nomodo - Manage your system by yourself

Autori (in ordine alfabetico): Giuseppe Glorioso Lucia Polizzi

June 9, 2018

Contents

1 Introduzione

1.1 Introduzione al progetto

Il progetto nomodo nasce dalla necessitá di un applicativo di gestione dei sistemi Ubuntu che sia più immediato ed accessibile rispetto al classico terminale, e quindi utilizzabile anche dagli utenti che per un motivo o per un altro non possono o non vogliono avere a che fare con il terminale. Nomodo si prende in carico di eseguire tutte le chiamate al terminale o meno per eseguire operazioni atte alla gestione del sistema presentando all'utente una interfaccia web chiara e comprensibile. Per operazioni in questo caso si intendono l'aggiornamento, la manutenzione e il miglioramento del sistema come ad esempio l'installazione dei pacchetti, la ricerca e la modifica dei file, così come operazioni di più alto livello come la gestione basilare del web server Apache.

1.2 Informazioni tecniche

Python + Flask L'applicativo scritto in python è basato sul framework Flask, utilizzato tra l'altro come webserver per l'accesso al pannello. Durante la fase di sviluppo si è utilizzato nginx come reverse proxy in modo da poter raggiungere il pannello web sulla porta 80 e non sulla 5000. È stata presa poi in seguito la decisione di lasciare che l'applicativo girasse sulla porta 5000 in quannto meno comune e quindi meno alla mercé degli hacker.

L'applicazione è stata quindi divisa in modo netto nelle due componenti fondamentali, il Frontend e il Backend che anche andremo quindi ad analizzare qui brevemente e più approfinditamente nei capitoli successivi:

- Il **backend** consiste in una serie di funzioni raccolte in una serie di file a mò di libreria, risiedenti nella cartella `systemcalls` (come ad es. `system.py` o `user.py`), utilizzati sia per la raccolta di dati sia per eseguire azioni sul sistema che non necessitano di output in uscita
- Il **frontend** rappresenta la parte grafica dell'applicativo web, e utilizza le funzioni del backend per la ricerca di informazioni e per la modifica alle componenti del sistema inclusa la modifica dei file quali i file di configurazioni

MongoDB Ogni operazione sensibile effettuata tramite l'applicazione comporta la memorizzazione delle modifiche che comporta la stessa in documento di mongodb, cosí da poter risalire alla storia delle operazioni effettuate e tentare un revert delle modifiche in caso ad esempio il sistema perda di stabilità o le modifiche non portino al risultato sperato. Tali operazioni possono riguardare ad esempio la modifica di un file, o la rimozione di un pacchetto dal sistema. Ogni log in mongodb presenta inoltre un flag **status** che indica se l'operazione eseguita sia andata a buon fine o meno, in modo da rendere più chiara la navigazione tra i log e dare la possibiltà all'utente di filtrarli in base a questo campo. [1]

2 Backend

Come anticipato in sezione 1.2 il backend è composto da una serie di funzioni raggruppate per categoria che fanno utilizzo di varie librerie python per compiere operazioni che possono o meno alterare lo stato del sistema. Allo stato attuale le categorie che compongono il backend sono le seguenti:

[1] Le uniche operazioni memorizzate nel database sono quelle relative all'utilizzo dell'applicativo; una modifica effettuata direttamente sul sistema ad esempio tramite il terminale va incontro alle regole del sistema Ubuntu e ogni modifica potrebbe essere irreversibile. In questi casi fare riferimento ai log del sistema che è possibile trovare al percorso `/var/log/` o sul pannello web alla sezione `Log`.

- Utenti
- Network
- Cron
- Sistema
- Apache
- Database
- File
- Logs

Interfaccia al frontend Ogni funzione chimata restituisce sempre un dizionario contenente almeno n codice di ritorno e il logid del documento inserito in mongo con un mongo `_id` se applicabile[2] oppure un logid `None` se non è stato creato alcun log. Distinguiamo quindi 2 casi in base al valore della variabile `returncode`:

- Se *returncode* = 0 l'operazione è andata a buon fine e il dizionario conterrà una terza variabile `data` che conterrà i dati richiesti se la funzione chiamata è tesa per restutuire output oppure sarà una variabile nulla se la funzione non restituisce output
- Se *returncode* $\neq$ 0 c'è stato un errore durante l'esecuzione dell'applicazione e il dizionario restituito conterrà quindi una terza variabile `stderr` il cui valore è un messaggio di errore e, se l'errore è dato da un comando eseguito in bash, il comando che una volta lanciato ha generato l'eccezione.

Il frontend o l'utente che voglia chiamare per qualsivoglia motivo le funzioni del backend direttamente, potrá farlo quindi nel seguente modo::

```
data = getifacestat()
if data['returncode'] is 0:
        data = data['data']
else:
        print( data['stderr'] )

pprint(data)
```

subprocess Le funzionalità di Python più utilizzata per la realizzazione dell'applicazione sono senza dubbio quelle appartenenti alla libreria subprocess, che permette di eseguire comandi come se si stessero eseguendo in bash. Si è cercato il più possibile di limitare l'utilizzo di questa libreria ma le sue funzionalità si sono rese necessarie nella maggior parte dei casi delle funzioni del backend, a causa dello scarsa agilità che ha python di interfacciarsi col sistema sottostante. In generale l'esecuzione di un comando avviene nel seguente modo:

[2] Cioè in caso l'operazione sia una operazione sensibile e richieda quindi un inserimento in mongo per tenere traccia della stessa

```
command = ['ifconfig', '-a']
try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True)
except CalledProcessError as e:
        return command_error(e, command, logid)

return command_success( output )
```

Tutte le funzioni di nomodo ritornano o con un `command_success` in caso l'operazione sia andata a buon fine o con un `command_error` in caso il comando non vada a buon fine e venga lanciata l'eccezione `CalledProcessError`. In entrambi i casi viene restituito il dizionario menzionato in sezione 2. Un esempio di comando che non restituisce output è il seguente:

```
def removeuser(user, removehome=None):

    logid = mongolog( locals(), getuser(user) )


    try:
        command = ['deluser', user]
        if removehome: command.append('--remove-home')

        check_output( command, stderr=PIPE, universal_newlines=True )
    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success(logid)
```

Nelle prossime sezioni verranno analizzate tutte le categorie e spiegato il funzionamento di ogni funzione che contengono.

2.1 utilities

Questa categoria contiene per la maggior parte funzioni che non vengono mai richiamate direttamente dal frontend, ma vengono utilizzate dalle altre funzioni del backend. Fa eccezione la funzione writefile che oltre essere chiamata da queste funzioni può anche essere chiamata direttamente, ed è utile alla modifica dei.
Analizziamo le funzioni di questa libreria nelle prossime sezioni.

2.1.1 mongolog()

```
def mongolog(params, *args):

    dblog = dict({
        'date': datetime.datetime.utcnow(),      #Operation date
        'funname': inspect.stack()[1][3],        #Function name
        'parameters': params,                    #Called function's
            parameters
```

```
    })

    for arg in args:
        dblog.update( arg )

    #ObjectID in mongodb
    return db.log.insert_one( dblog ).inserted_id
```

Viene chiamata ogni volta che una funzione sia classificata come **sensibile** cioè che va a modificare lievemente o pesantemente il sistema e prende in carico di creare un log mongodb contenente le operazioni eseguite e i dati modificati dalla funzione. Accetta N parametri di cui il primo (obbligatorio) è la lista di parametri con cui è stata lanciata la funzione di cui si sta memorizzando il log. Ad esempio in

```
def ifacedown( iface ):
        logid = mongolog( locals() )
        ...
```

il primo parametro è `locals()` che contiene la variabile `iface` che verrà quindi memorizzata nel log di mongo; il secondo parametro (opzionale) può essere uno o più dizionari da unire al dizionario memorizzato in mongodb. Ad esempio nella funzionei `addusertogroups`:

```
def addusertogroups(user, *groups):

    #Logging operation to mongo first
    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )
    ...
```

Si è deciso che prima di aggiungere un utente a dei nuovi gruppi si va a memorizzare in mongo non solo `locals()` e quindi `user` e `*groups` ma anche le informazioni sull'utente ricavate attraverso la funzione `getuser()` e passate a `mongolog()` come secondo parametro.

Il dizionario di base memorizzato in mongo è formato da tre elementi:

- La data in cui viene effettuata l'operazione
- Il nome della funzione che ha chiamato `mongolog`, ricavata tramite il supporto della libreria `inspect`
- I parametri della funzione che chiama, come spiegato in precedenza, e ottenuti chiamando la funzione `locals()`

2.1.2 mongologstatus() e funzioni collegate

```
def mongologstatus(logid, status):
```

```
    return db.log.update_one(
        {'_id': logid},
        {'$set': { 'status' : status }},
        upsert=False
        )

def mongologstatuserr(logid, status='error'):
    return mongologstatus(logid, status)
def mongologstatussuc(logid, status='success'):
    return mongologstatus(logid, status)
```

Parametri Accetta 2 parametri:

- `logid`: è il logid del documento di mongo a cui aggiungere o modificare il campo status
- `status='error'`: è lo stato da assegnare la log individuato da `logid`

Questa funzione è intesa per aggiungere o modificare il campo `status` di un log di MongoDB. Le funzioni di nomodo (come è giusto che sia) creano un documento di mongo per memorizzare le informazioni sull'operazione prima di procedere all'operazione stessa. In caso un'operazione non andasse nel modo aspettato bisognerebbe quindi marcare il documento appena creato in mongo in modo da avvisare l'utente che sta consultando il log che l'operazione riferita a quel documento non è andata a buon fine. La memorizzazione del log avviene quindi nei seguenti step:

1. Viene lanciata la funzione che richiede la memorizzazione del log e quindi mongolog(), che va a creare il log senza nessuna indicazione sul successo o meno dell'operazione
2. Dopo l'esecuzione della funzione viene chiamata `command_success` se l'operazione è andata a buon fine; la prima operazione che questa va ad eseguire è chiamare a sua volta la funzione `mongologstatussuc()` che chiama `mongologstatus()` con il parametro `status='error'` aggiungendo tale campo `status` al log di mongo ed indicando la buona riuscita dell'applicazione all'utente che andrà ad analizzare i log
3. In caso invece la funzione vada in errore viene chiamata command_success che chiama `mongologstatuserr()` che chiama `mongologstatus()` con il secondo parametro `status='error'` aggiungendo tale campo al log di mongo
4. In caso invece si voglia personalizzare il campo `status` basta quindi che la funzione chiami direttamente `mongologstatus()` con il secondo parametro `status` ad un qualsivoglia valore si voglia inserire, ad es. `status='canceled'`

Si intuisce quindi da questi step che un log che non abbia il campo status indica un crash della funzione nel codice che è intercorso tra la memorizzazione del log e l'aggiunta del campo `status`.

L'operazione deve fallire se il documento indicato da `logid` non esiste, quindi si è aggiunta la direttiva `upsert=False`.

Ecco un esempio che mostra lo stato di un log appena aggiunto (senza il campo `status`) e al termine dopo aver chiamato `command_success`:

```
> db.log.find()
{ "_id" : ObjectId("5ae596d4bf3bd205c1aeea25"), "parameters" : { "shell"
   : "/bin/bash", "user" : "giuseppe2", "password" : "test" }, "funname"
   : "adduser", "date" : ISODate("2018-04-29T09:56:36.627Z") }
> db.log.find()
{ "_id" : ObjectId("5ae596d4bf3bd205c1aeea25"), "parameters" : { "shell"
   : "/bin/bash", "user" : "giuseppe2", "password" : "test" }, "funname"
   : "adduser", "date" : ISODate("2018-04-29T09:56:36.627Z"), "status" :
   "success" }
```

Return Restituisce un oggetto della classe

2.1.3 command_success

```
def command_success( data=None, logid=None, returncode=0 ):

    if logid:
        mongologstatussuc( logid )

    return dict({
        'returncode': returncode,
        'data': data,
        'logid': logid
    })
```

La funzione `command_success` fondamentalmente costruisce il dizionario da restituire all'utente quando una funzione del backend ha finito le sue opoerazioni e non ci sono stati errori durante l'esecuzione. Insieme alla sorella command_error sono le uniche due funzioni chiamate al termine di una funzione del backend.

Parametri Accetta tre parametri:

- `data=None`: Sono i dati da restituire all'utente se la funzione che l'ha chiamata li genera. Di base è `None`
- `logid`: Il logid a cui aggiungere il campo `status` e da restituire all'utente nel dizionario come campo del dizionario. Di base è `None` in quanto il chiamante potrebbe non aver generato un mongolog per l'operazione che ha effettuato
- `returncode`: È il codice di ritorno che verrà inserito nel dizionario. Essendo questa funzione invocata ogni qualvolta il chiamante esegue tutte le operazioni senza errore di base questo parametro è 0 ad indicare successo e può quindi essere omesso, ma può essere personalizzato passandolo alla chiamata

Funzionamento La prima operazione eseguita è la chiamata a`mongologstatussuc()` per aggiungere al log di mongo il campo `status`, questo solo in caso il parametro `logid` sia non nullo e quindi la funzione chiamante ha dovuto memorizzare un mongolog. Successivamente va a costruire il dizionario da restituire formato dal codice di ritorno, i dati voluti dall'utente (se disponibili, altrimenti `None`), e il *logid* dell'operazione.

Return Restituisce il dizionario contenente i parametri passati alla funzione o i loro valori di default se non vengono passati. Da utilizzare come spiegato in sezione 2.

2.1.4 command_error

```
def command_error( e=None, command=[], logid=None, returncode=1, stderr='
    No messages defined for this error' ):

    if logid:
        mongologstatuserr( logid )

    return dict({
        'returncode': e.returncode if e else returncode,
        'command': ' '.join(command),
        'stderr': e.stderr if e else stderr,
        'logid': logid
    })
```

`command_error` è l'opposto di `command_success`. Come visto in sezione 2 viene invocato quando un comando lanciato attraverso la libreria `subprocess` fallisce nell'esecuzione. Può essere però anche usato per generare un dizionario di errore personalizzato.

Parametri Accetta 5 parametri:

- `e=None`: è l'oggetto creato nel caso in cui venga lanciata l'eccezione `CalledProcessError`
- `command=[]`: è il comando la cui esecuzione ha generato l'eccezione. Lista vuota di default
- `logid`: Il logid a cui aggiungere il campo `status` e da restituire all'utente nel dizionario come campo del dizionario. Di base è `None` in quanto il chiamante potrebbe non aver generato un mongolog per l'operazione che ha effettuato
- `returncode=1`: Un codice di ritorno personalizzato da inserire nel dizionario in caso il parametro `e` sia nullo
- `stderr='No messages defined for this error'`: È il messaggio di errore da inserire nel dizionario in caso il parametro `e` sia nullo

Funzionamento La funzione costruisce un dizionario da restituire all'utente contenente le varie informazioni sull'errore che è accaduto, ossia il codice di ritrorno, il messaggio di errore, il comando che ha generato l'errore (da passare in input) e il logid del documento in mongo che riguarda il comando/operazione.

Distinguiamo 3 casi:

- Viene generato un oggetto del tipo `CalledProcessError` appartenente a subprocess: in questo caso si passa alla funzione l'oggetto generato e il comando che ha causato l'errore. La funzione ricava automaticamente da questo oggetto il codice e il messaggio di erroree lo inserisce nel dizionario insieme al comando di `subprocess` che ha causato l'errore. È quindi in questo caso necessario passare almeno questo oggetto e il comando (che non è però strettamente necessario)

- Si vuole generare un errore personalizzato: in questo caso invece i parametri `e` e `command` non devono essere passati, e al loro posto vengono passati `returncode` e `stderr` che verranno inseriti nel dizionario da restituire.
- Si vuole generare un errore di default: in quest'ultimo caso basta chiamare la funzione senza passare alcun parametro e viene generato un errore di base in cui i valori del codice di ritorno saranno quelli assegnati di default e che si può vedere nella sezione *Parameters*

Oltre a restituire il dizionario di errore questa funzione, se il parametro `logid` è non nullo agisce come command_success aggiungendo quindi al log dell'operazione il campo `status` col valore `error`.

Return Restituisce il dizionario creato come descritto nel funzionamento e come si può vedere nel codice.

2.1.5 writefile

```
def writefile(filepath, newcontent=None, force=False):

    if not newcontent:
        try:
            with open(filepath, 'r') as content:
                return content.read()
        except FileNotFoundError:
            return command_error( returncode=10, stderr='No file found on
                path : "'+filepath+'"' )

    if not force:
        md5new = hashlib.md5()

        md5new.update( newcontent.encode() )
        md5new = md5new.hexdigest()

        md5old = hashlib.md5( open( filepath, 'rb' ).read() ).hexdigest()

        if md5new == md5old:
            return command_error( returncode=2, stderr='Nothing to write(
                no changes from original file). You can force writing
                using the parameter "force=True"' )

    localsvar = locals()
    del localsvar['newcontent']
    logid = mongolog( localsvar, filediff(filepath, newcontent) )

    opened = open(filepath, 'w')
    opened.write(newcontent)
    opened.close()

    return command_success( logid=logid )
```

Questa funzione è intesa per dare un supporto in caso l'utente voglia modificare un file. Implementa tutte le operazioni e i controlli che bisognerebbe effettuare prima della modifica di un file, incluso un mongolog da cui si possa risalire al contenuto del file prima della scrittura. È inoltre capace di restituire il contenuto di un file, utile in fase di pre-scrittura.

Parametri Sono 3 i paramentri accettati da questa funzione:

- `filename`: è il percorso del file sul sistema che si vuole modificare
- `towrite=None`: è il nuovo contenuto del file da scrivere sullo stesso
- `force=False`: questa variabile (di base a `False`) se `True` forza la scrittura del nuovo contenuto anche se questo non differisce dal contenuto originale del file

Funzionamento

```
    if not newcontent:
        try:
            with open(filepath, 'r') as content:
                return content.read()
        except FileNotFoundError:
            return command_error( returncode=10, stderr='No file found on
                path :"'+filepath+'"' )
```

Oltre a scrivere i file la funzione può essere utilizzata per restituire all'utente o al frontend il contenuto di un file. Come si vede dal primo `if` se il parametro `newcontent` è nullo semplicemente il file viene aperto in lettura e ne viene restituito il suo contenuto. Questa variabile dovrebbe contenere il nuovo contenuto del file.

```
    if not force:
        md5new = hashlib.md5()

        md5new.update( towrite.encode() )
        md5new = md5new.hexdigest()

        md5old = hashlib.md5( open( filename, 'rb' ).read() ).hexdigest()

        if md5new == md5old:
            return command_error( returncode=2, stderr='Nothing to write(
                no changes from original file). You can force writing
                using the parameter "force=True"' )
```

In questa seconda parte controlla la varibile `force`. Se questa è `false` genera l'md5 del nuovo contenuto (towrite) e del vecchio contenuto (quello del file) e nell'ultimo `if` ne controlla l'uguaglianza. Quindi se `force=False` e `md5new==md5old` non c'è necessità di scrivere il file e genera quindi un messaggio di errore personalizzato (o potremmo dire di warning in questo caso) usando la funzione command_error, e che avverte il chiamante (o il fontend) che non è stata eseguita alcuna operazione ma che la si può forzare passando il parametro `force` col valore booleano `True`.

```
    localsvar = locals()
    del localsvar['towrite']
    logid = mongolog( localsvar, filediff(filename, towrite) )

    #Writing new content to "filename" file
    opened = open(filename, 'w')
    opened.write(towrite)
    opened.close()

    return command_success( logid=logid )
```

Si arriva quindi a questo pezzo di codice se `force=True` o se `force=False` e `md5new!=md5old`. Questa funzione andando a modifcare file che possono essere o meno importanti necessità di un log che memorizzi una quantità di informazioni tali da consentire il ripristino del file vecchio. Si intuisce però che memorizzare interamente il vecchio e il nuovo contenuto sarebbe impensabile per avere un mongolog decente e di piccole dimensioni. Si è quindi optato per inserire il solo diff tra vecchio e nuovo contenuto. In casi comuni e per i nostri scopi questa scelta porta ad una riduzione significativa della dimensione del log, in quanto raramemnte un utente cancella completamente e riscrive file di migliaia di righe di codice.
Passando al funzionamento quindi si memorizza dapprima ciò che restituisce la funzione `locals()` (che come spiegato restituisce i parametri con cui la funzione è stata chiamata) e si inserisce ciò che ritora nella variabile `localsvar`. Dopodichè da questa variabile (che è diventata un dizionario dopo l'assegnazione) si elimina il parametro `towrite` che a quanto ne sappiamo potrebbe anche essere di migliaia di righe. Si crea a questo punto un mongolog che non abbia più al suo interno il contenuto scritto ma ciò che ritorna la funzione filediff che come vedremo in seguito non fa altro che restituire il diff tra il vecchio e il nuovo contenuto del file. Una volta effettuato i controlli e memorizzati in mongo i dati necessari si procede finalmente a scrivere il nuovo contenuto sul file tramite le funzioni Python per la gestione dei file. Si invoca infine `command_success` passandogli il `logid`.

Return Restituisce il *dizionario di successo* generato dalla funzione `command_success` con all'interno il solo `logid` dell'operazione.

2.1.6 filediff

```
def filediff(filea, fileb):
    if not os.path.exists(filea):
        filecontent = filea
        filea = '/tmp/.nomodotempa'
        with open(filea, 'w') as opened:
            opened.write(filecontent)

    if not os.path.exists(fileb):
        filecontent = fileb
        fileb = '/tmp/.nomodotempb'
        with open(fileb, 'w') as opened:
            opened.write(filecontent)
```

```
    command = ['diff', filea, fileb]

    output = Popen(command, stdout=PIPE, universal_newlines=True).
        communicate()[0]

    return {'filediff': output }
```

La funzione `filediff` come si intusice dal nome serve a generare un diff tra 2 file. È usata principalmente dalla funzione writefile per le sue operazioni di log su mongo ma può anche essere chiamata direttamente dall'utente curioso o dal fontend.

Parametri Accetta 2 parametri. Questi non si differenziano l'uno dall'altro e possono essere un percorso ad un file sul sistema, o una stringa. È perfettamente legale che uno sia di un tipo e uno di un altro, ad es. `filea` può essere il percorso di un file mentre `fileb` una stringa che rappresenta il contenuto di un file, così come accade quando questa funzione viene chiamata da `writefile`.

Funzionamento Innanzitutto nei primi due `if` la funzione controlla se i parametri contengono una stringa che identifica il percorso di un file sul sistema o un contenuto usando la libreria `os`. Se uno dei due parametri non contiene un percorso si prende in carico di scrivere la stringa che contiene in un file temporaneo creato al momento. Ad es. se `filea` non è un file ma una stringa crea un nuovo file vuoto, `/tmp/.nomodotempa` in questo caso, e ci scrive il contenuto di `filea`.
Nella seconda parte viene composto e lanciato il comando `Popen` di `subprocess` [3] per effettuare il diff in linux style. Si è optato per tale diff in quanto nessuna funzione di Python restituisce il diff così chiaramente e in modo così adatto da essere inserito in un database.

Return Restituisce un dizionario con la sola chiave `filediff` dove il valore è il diff generato.

2.1.7 delfile

```
    logid = mongolog( locals() )

    try:
        os.remove( path )
    except FileNotFoundError:
        return command_error( returncode=10, stderr='File to remove not
            found: "'+path+'"' )

    return command_success( logid=logid )
```

Nasce per eliminare un file dal sistema.

Parametri Accetta un solo parametro che è il percorso del file da rimuovere.

[3] Da notare che questa è una delle poche funzioni che usa `Popen` al posto di `check_output` e `check_call`. Questo è dovuto al fatto che i codici di ritorno del comando `diff` sono diversi da 0 se esistono differenze tra i 2 file analizzati, e utilizzare le due funzioni menzionate genererebbe un eccezione `CalledProcessError` che farebbe crashare il programma

Funzionamento Semplicemente eliminare il file utilizzando la funzione `remove` della libreria `os`. Durante la rimozione verifica che non si verifichi l'eccezione `FileNotFoundError`, lanciata se il file indicato dal percorso non esiste. In caso l'eccezione venga lanciata viene chiamata la funzione di errore con codice e messaggio di errore personalizzati.

Return Restituiusce il dizionario di errore costruita come si vede in codice se il file non esiste, il dizionario di successo altrimenti, contenente nella variabile `logid` l'object id del documento mongo creato e contenente le informazioni sull'operazione appena eseguita.

2.2 Utenti

2.2.1 getuser

```
def getuser(user):

    try:
        command = ['getent', 'passwd',  user]
        userinfo = check_output(command, stderr=PIPE, universal_newlines=
           True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )


    #Info sull'utente dal file /etc/passwd
    userinfo = userinfo[0].split(':')


    #Getting user groups
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups #Returns the entire error dictionary as created
            by "command_error" function


    return command_success( data=dict({
        'uname': userinfo[0],
        'canlogin': 'yes' if userinfo[1]=='x' else 'no',
        'uid': userinfo[2],
        'gid': userinfo[3],
        'geco': userinfo[4].split(','),
        'home': userinfo[5],
        'shell': userinfo[6],
        'group': usergroups.pop(0), #Main user group
        'groups': usergroups if usergroups else "<␣No␣groups␣>"
    }) )
```

Questa funzione restituisce tutti le informazioni di un utente così come lette da comando `getent` e quindi dal file `/etc/passwd`.

Parametri Accetta l'unico `user` che è il nome utente dell'utente di cuisi vogliono le informazioni. Questo utente deve esistere nel sistema e deve quindi essere presente nel file `/etc/passwd`. È possibile ottenere la lista di utenti che possono essere usati per questa funzione leggendo i valori del dizionario restituito dalla funzione getusers.

Funzionamento

```
    try:
        command = ['getent', 'passwd',  user]
        userinfo = check_output(command, stderr=PIPE, universal_newlines=
           True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )

        userinfo = userinfo[0].split(':')
```

Innanzitutto viene costruito e lanciato il comando `getent passwd <user>` che ricava le informazioni sull'utente dal file `/etc/passwd`. L'esecuzione viene controllata per catturare un eventuale eccezione `CalledProcessError`. Da notare che l'output del comando viene diviso per righe dalla funzione `splitlines()` posta alla fine di `check_output()`.
Dato che le informazioni sull'utente sono divise da un due punti ma racchiusi in una stringa queste vengono separate dalla funzione `split()` che crea una lista di stringhe.

```
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups #Returns the entire error dictionary as created
            by "command_error" function
```

Visto che `getent` non restituisce la lista dei gruppo di cui l'utente fa parte ma solo il principale, ricaviamo questa lista dalla funzione getusergroups, facendo gli opportuni controlli sul codice di ritorno come spiegato in sezione 2.

```
return command_success( data=dict({
        'uname': userinfo[0],
        'canlogin': 'yes' if userinfo[1]=='x' else 'no',
        'uid': userinfo[2],
        'gid': userinfo[3],
        'geco': userinfo[4].split(','),
        'home': userinfo[5],
        'shell': userinfo[6],
        'group': usergroups.pop(0), #Main user group
        'groups': usergroups if usergroups else "< No groups >"
}) )
```

Non resta quindi che creare un dizinario da dare in pasto a `command_success` per essere restituita agli utenti. A parte i campi che vengono inseriti normalmente ne distinguiamo quattro che si comportano in modo diverso:

- `canlogin`: indica se è possibile effettuare l'accesso alla shell con l'utente. In particolare se il campo di `/etc/passwd` è `x` allora l'utente può effettuare l'accesso
- `geco`: Indica l'anagrafica dell'utente e altre informazioni come l'email. Questo campo è una lista ma presentatdosi come una semplice stringa divisa da virgole necessita di essere splittata prima dell'inserimento in modo da poter essere riferita direttamente
- `group`: È il gruppo principale dell'utente e viene ricavata dalla lista di gruppi in quanto primo membro, e poi rimosso da questa lista
- `groups`: È la lista dei gruppi secondari di cui l'utente fa parte e viene inserita così com'è (una lista) se il l'utente ha almeno un gruppo secondario, altrimenti viene inserita una stringa che indica l'assenza dei gruppi in modo da non vedere apparire in questo campo una lista vuota

Return Il dizionario descritto nel funzionamento, composto quindi dai seguenti campi:

- uname: Nome utente
- canlogin: Indica la possibilità di accesso alla shell con questo utente
- uid: L'user ID dell'utente[4]
- gid: il group ID del gruppo principale di cui l'utente fa parte, che di base ha lo stesso nome dell'utente
- geco: Alcune informazioni sull'utente, ossia nome, cognome, email, stanza ecc.
- home: la home dell'utente; solitamente se l'utente non è di sistema si trova al percorso `/home/<uname>`
- shell: la shell assegnata all'utente. Di base è `/bin/bash` ma solitamente se l'utente non è di sistema si possono trovare le shell fittizie `/bin/false` e `/usr/bin/nologin`
- group: il nome del gruppo principale di cui l'utente fa parte. Solitamente alla creazione dell'utente viene creato dal sistema anche questo gruppo e gli viene dato lo stesso nome. Ad es. l'utente *giuseppe* ha come gruppo principale *giuseppe* e di base è l'unico membro
- groups: la lista dei gruppi secondari di cui l'utente faf parte. È possibile in nimodo aggiungere un utente ad uno o più gruppi usando la funzione addusertogroups

2.2.2 getusers

```
def getusers():

    with open('/etc/passwd', 'r') as opened:
        passwd = opened.read().splitlines()


    users = dict()
    for line in passwd:
        line = line.split(':', 3)
        uname = line[0]
```

[4]Nei sistemi UNIX based gli utenti non di sistema hanno un uid che parte da 1000 a salire.

```
        uid = line[2]
        users[uid] = uname

    return command_success( data=users )
```

Questa funzione nasce per ottenere la lista utenti presente nel sistema.

Parametri La funzione non prende parametri

Funzionamento

```
    with open('/etc/passwd', 'r') as opened:
        passwd = opened.read().splitlines()
```

Siccome si cerca di limitare il più possobile l'utilizzo delle funzioni della libreria `subprocess` gli utenti vengono letti dal file `/etc/passwd` con questa open; le righe di questo file vengono quindi divise ed inserite nella variabile `passwd` che diventa quindi una lista di stringhe.

```
    users = dict()
    for line in passwd:
        line = line.split(':', 3)
        uname = line[0]
        uid = line[2]
        users[uid] = uname

    return command_success( data=users )
```

Eseguito questo passaggio si itera sulle linee del dizionario `passwd`; ogni riga viene splittata di tre elementi in quanto `uname` si trova nel primo campo mentre `uid` si trova nel terzo, e vengono inserite in dizionario in cui la chiave è l'uid mentre il valore è lo uname. Questo è il dizionario da essere restituito all'utente, che viene quindi passato alla funzione di uscita `command_success`.

Return Restituisce il dizionario costruito in cui per ogni utente la chiave è l'uid mentre il valore è lo uname.

2.2.3 getgroups

```
def getgroups(namesonly=False):

    with open('/etc/group', 'r') as opened:
        etcgroup = opened.read().splitlines()


    groups = list()
    if namesonly:
        groups = list(map( lambda line: line.split(':')[0], etcgroup ))
    else:
        for line in etcgroup:
            line = line.split(':')
            groups.append({
                'gname': line[0],
```

```
                'gid': line[2],
                'members': line[3].split(',')
            })

    return command_success( data=groups )
```

Questa funzione agisce nello stesso modo di `getusers()` restituendo però i gruppi del sistema invece che gli utenti.

Parametri La funzione accetta un unico parametro `namesonly` che di base è `False` e che se impostato a `True` restutuisce il solo nome dei gruppi.

Funzionamento Agisce come getusers e cioè inizialmente legge la lista dei gruppi dal file `/etc/group`, lo divide nelle sue righe e lo memorizza nella variabile `etcgroup` che diventa quindi una lista di stringhe.

```
    groups = list()
    if namesonly:
        groups = list(map( lambda line: line.split(':')[0], etcgroup ))
```

Eseguito questo passaggio dichiara la lista `groups` che andrà ritornata e verifica il valore della variabile `namesonly`. Se questo è `True` applica una funzione inline `lambda` e per ogni elemento della lista creata applica uno split dei suoi campi e ne memorizza il primo valore nella lista da restituire `groups`. In questo modo quindi `groups` sarà una lista di stringhe in cui ogni stringa è il nome di un gruppo.

```
else:
    for line in etcgroup:
        line = line.split(':')
        groups.append({
            'gname': line[0],
            'gid': line[2],
            'members': line[3].split(',')
        })

return command_success( data=groups )
```

Se invece `namesonly` è `False` si entra in un for in cui per ogni riga del dizionario `etcgroup`:

1. la riga stessa viene splittata nei suoi campi che vengono memorizzati nella variabile `line`
2. Viene creato un dizionario che contiene nome, identificativo, e membri appartenenti al gruppo
3. La lista di dizionari creata (`groups`) viene passata alla funzione di ritorno `command_success`

Return Restituisce il dizionario costruito che contiene, per ogni gruppo del file `/etc/group`:

- gname: il nome del gruppo
- gid: l'identificativo del gruppo
- members: una lista di stringhe che contiene tutti i membri del gruppo. Notare che nel codice questa lista è stata creata splittando il componente numero 3 della riga per la virgola

2.2.4 getusergroups

```
def getusergroups(user):

    command = ['groups', user]

    try:
        usergroups = check_output(command, stderr=PIPE,
            universal_newlines=True).splitlines()
    except CalledProcessError as e:
        command_error( e, command )

#                   .------Removing username from list
#                   |                          .----------.
#                   V                          V          |
    usergroups = re.sub('^.*: ', '', usergroups[0]) #Only first line
        contains the groups
    usergroups = usergroups.split(' ')

    return command_success( data=usergroups )
```

Restituisce la lista di gruppi di cui l'utente fa parte.

Parametri L'unico parametro che prende è `user` che è il nome utente di cui si vogliono conoscere i gruppi.

Funzionamento

```
    command = ['groups', user]

    try:
        usergroups = check_output(command, stderr=PIPE,
            universal_newlines=True).splitlines()
    except CalledProcessError as e:
        command_error( e, command )
```

In questa prima parte viene composto e lanciato il comando `groups` che restituisce la lista di gruppi di cui l'utente fa parte. Con la funzione `splitlines()` viene poi diviso l'output in righe in quanto nell'infausto caso il comando genera più righe a noi server solamente la prima.
Viene generata così una lista di dizionari che viene memorizzata in `usergroups`.
Siccome a noi interessa l'output del comando viene lanciata la funzione `check_output` invece di `check_call`.
Anche in questo caso viene controllato che non venga generata una eccezione `CalledProcessError`.

```
    usergroups = re.sub('^.*: ', '', usergroups[0]) #Only first line
        contains the groups
    usergroups = usergroups.split(' ')

    return command_success( data=usergroups )
```

L'output di `groups` include prima della lista lo username dell'utente, quindi prima di proseguire deve essere rimosso. Questo viene fatto con la libreria sub che con un espressione regolare elimina tutto

quello che c'è prima e lo spazio che c'è dopo il due punti.
Fatta questa operazione basta quindi splittare la stringa restituita che divide i gruppi con uno spazio e chiamare la funzione `command_success` passadogli la lista di gruppi

Return Una lista di stringhe in cui ogni stringa in cui ogni stringa è un gruppo di cui l'utente `user` fa parte.

2.2.5 getusernotgroups

```
def getusernotgroups(user):

    #Getting all system groups
    groups = getgroups(namesonly=True)
    if groups['returncode'] is 0:
        groups = groups['data']
    else:
        return groups

    #Getting user specific groups
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups

    usernotgroup = list(filter( lambda group: not any(s in group for s in
        usergroups), groups ))

    return command_success( data=usernotgroup )
```

Nasce per completare la funzione getusergroups e al contrario di questa restituisce tutti i gruppi di cui l'utente non fa parte.
A cosa può mai servire questa funzione? Durante l'aggiunta di un utente ad un gruppo si deve avere la necessità di sapere l'utente di quali gruppi fa parte e di quali non fa part per rendere l'interfaccia può chiara ed agevole, più facile da usare e anche per garantire il corretto funzionamento dell'applicativo, riducendo le situazioni di eccezione in caso di situazioni che non avevamo programmato.

Parametri Prende l'unico parametro `user` che è il nome utente da usare per ricavare la lista dei gruppi di cui l'utente stesso non fa parte.

Funzionamento

```
    #Getting all system groups
    groups = getgroups(namesonly=True)
    if groups['returncode'] is 0:
        groups = groups['data']
    else:
        return groups
```

```
    #Getting user specific groups
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups
```

Per ricavare la lista di gruppi di cui l'utente non fa parte si è esegue la sottrazione tra tutti i gruppi del sistema e tutti i gruppi di cui l'utente fa parte.
In questa prima parte quindi chiamando le funzioni `getgroups()` e `getusergroups()` si ricavano rispettivamente tutti i gruppi di sistema e tutti i gruppi di cui l'utente fa parte. Sul dizionario restituito si effettuano delle verifiche per vedere se l'operazione è andata a buon fine.

```
    usernotgroup = list(filter( lambda group: not any(s in group for s in
        usergroups), groups ))

    return command_success( data=usernotgroup )
```

Per la sottrazione utilizziamo la funzione `filter`. Questa funzione valuta l'espressione che gli si da, se questa restituisce `True` allora inserisce l'oggetto nella lista che sta costruendo, altrimenti lo omette. Ad es. in questo caso per ogni `group` oggetto di `groups` e per ogni gruppo `s` in `usergroups` se `s` è uguale a `group` (si è usato `in` in questo caso) viene restituito `True` che viene negato a `False` e quindi il `group` non viene inserito nella lista che la funzione sta costruendo.
Le funzioni usate sono quindi le seguenti:

- `ffilter`: filtra i risultati in base all'espressione che gli si da. Se `True` li inserisce nella lista che sta costruendo, altrimenti li omette
- `any`: restituisce `True` se almeno una delle espressioni al suo interno restituisce `True`. In questo caso se almeno uno dei gruppi dell'utente coincide col gruppo `s`. Questa espressione viene negata usando `not`, quindi si ricavano tutti i gruppi che sono nella lista dei gruppi di sistema `groups` ma non nella lista dei gruppi dell'utente

Una volta effettuata la sottrazione chiama la funzione di ritorno che indica *successo* `command_success` passandogli la lista dei gruppi ricavati.

Return Restituisce una lista di stringhe in cui ogni stringa è il nome di un gruppo di cui l'utente non fa parte.

2.2.6 addusertogroups

```
def addusertogroups(user, *groups):

    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo
```

```
    logid = mongolog( locals(), userinfo )

    try:
        for group in groups:
            command = ['adduser', user, group],
            check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

La funzione nasce per aggiungere un utente ad uno o più gruppi di sistema. Ovviamente l'utente non deve appartenere al/ai gruppo/i a cui si sta aggiungendo. È possibile ricavare una lista di questi gruppi utilizzando la funzione getusernotgroups.

Parametri

- `user`: la prima non poteva che essere l'utente che si vuole aggiungere ai gruppi
- `*groups`: la seconda è una variabile accumulativa di Python, ciò significa che dopo aver passato `user` si possono passare quanti gruppo si vuole e verranno sempre unificati in questa variabile, per poi essere scorsi un pò alla volta. Questo ci da la possibilità di poter aggiungere l'utente a quanti gruppi si vuole chiamando questa funzione una sola volta

Funzionamento

```
    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )
```

Essendo questa una opoerazione sensibile che va a modificare una parte del sistema si va prima di tutto a memorizzare un mongolog per tenere traccia dell'operazione. In questo caso oltre a memorizzare i parametri con cui è stata chiamata (ricavati usando la funzione `locals()`) si ricavano le informazioni sull'utente chiamando la funzione getuser e si inseriscono nel log.

```
    try:
        for group in groups:
            command = ['adduser', user, group],
            check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

L'aggiunta dell'utente ai gruppi avviene in un ciclo `for` dove per ogni gruppo passato a `*groups` viene lanciato il comando `adduser` tramite una `check_call`. Notare che in questo caso non ci interessa

l'output del comando ma solo il suo codice di ritorno, è per questo che usiamo `check_call`. Se l'operazione va in eccezione come sempre di chiama la funzione `command_error` mentre se va a buon fine si chiama `command_success`, questa volta passandogli il solo logid. Il dizionario restituito all'utente avrà quindi la variabile `Data` a `None`, mentre la variabile `logid` conterrà il l'object id dell'operazione.

Return Non restituisce niente in output, la variabile `logid` invece sarà allocata e conterrà l'object id dell'operazione effettuata.

2.2.7 removeuserfromgroups

```
def removeuserfromgroups(user, *groups):

    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )

    try:
        for group in groups:
                command = ['gpasswd', '-d', user, group]
                check_call(command)
    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success( logid=logid )
```

Questa funzione funzione esattamente al contrario di addusertogroups, ossia permette di specificare uno i più gruppi da cui l'utente deve essere rimosso. È possibile ricavare una lista di questi gruppi utilizzando la funzione getusergroups.

Parametri

- `user`: l'utente che si vuole aggiungere ai gruppi
- `*groups`: è una variabile accumulativa di Python, ciò significa che dopo aver passato `user` si possono passare quanti gruppo si vuole e verranno sempre unificati in questa variabile, per poi essere scorsi un pò alla volta. Questo ci da la possibilità di poter rimuovere l'utente da quanti gruppi si vuole chiamando questa funzione una sola volta

Funzionamento

```
    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo
```

```
    logid = mongolog( locals(), userinfo )
```

Essendo questa una opoerazione sensibile che va a modificare una parte del sistema si va prima di tutto a memorizzare un mongolog per tenere traccia dell'operazione. In questo caso oltre a memorizzare i parametri con cui è stata chiamata (ricavati usando la funzione `locals()`) si ricavano le informazioni sull'utente chiamando la funzione getuser e si inseriscono nel log.

```
    try:
        for group in groups:
                command = ['gpasswd', '-d', user, group]
                check_call(command)
    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success( logid=logid )
```

La rimozione dell'utente dai gruppi avviene in un ciclo `for` dove per ogni gruppo passato a `*groups` viene lanciato il comando `gpasswd` tramite una `check_call`. Notare che in questo caso non ci interessa l'output del comando ma solo il suo codice di ritorno, è per questo che usiamo `check_call`. Se l'operazione va in eccezione come sempre di chiama la funzione `command_error` mentre se va a buon fine si chiama `command_success`, questa volta passandogli il solo logid. Il dizionario restituito all'utente avrà quindi la variabile `Data` a `None`, mentre la variabile `logid` conterrà il l'object id dell'operazione.

Return Non restituisce niente in output, la variabile `logid` invece sarà allocata e conterrà l'object id dell'operazione effettuata.

2.2.8 updatepass

```
def updateuserpass(user, password):

    localsvar = locals()
    del localsvar['password']
    logid = mongolog( localsvar )

    try:
        command = ['echo', user + ':' + password]
        p1 = Popen(command, stdout=PIPE)
        command = ['/usr/sbin/chpasswd']
        p2 = Popen(command, stdin=p1.stdout)
        p1.stdout.close()

    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success( logid=logid )
```

La funzione nasce per aggiornare la password dell'utente indicato.

Parametri

- `user`: Il nome utente dell'utente a cui cambiare la password
- `password`: La nuova password dell'utente. Notare che non serve la vecchia password in quanto l'applilcativo opera con privilegi di root

Funzionamento

```
    localsvar = locals()
    del localsvar['password']
    logid = mongolog( localsvar )
```

Essendo questa una operazione sensibile si deve prima di tutto creare un mongolog per tenere traccia dell'operazione. In questo caso però non possiamo memorizzare il parametro `password` in mongolog ne in chiaro e ne criptato in quanto sui sistemi unix-based anche se l'utenza root ha i privilegi per modificare le password di tutti gli utenti non può e non deve conoscere le password di questi. Quindi come prima cosa eliminiamo si crea una lista con i parametri chiamando la funzione `locals()` e poi si crea il mongolog.

```
    try:
        command = ['echo', user + ':' + password]
        p1 = Popen(command, stdout=PIPE)
        command = ['/usr/sbin/chpasswd']
        p2 = Popen(command, stdin=p1.stdout)
        p1.stdout.close()

    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success( logid=logid )
```

Dopo la memorizzazione del momgolog si passa all'esecuzione dei comando necessari. Il comando per cambiare la password di un utente in modo non interattivo prevede l'utilizzo di una pipe. Con la prima esecuzione quindi si stampa sullo STDOUT la stringa `<username>:<password>'`, tale stringa viene poi catturata dal comando `/usr/sbin/chpasswd` tramite la direttiva `stdin=p1.stdout` che la parsa e cambia la password dell'utente.
Se si va in eccezione (ad esempio si passa un utente non esistente) viene creato il dizionario di command_error e resttuito all'utente, mentre se l'operazione termina correttamemnte viene creato il dizionario di command_success e restituito all'utente.

Return Viene restituito il dizionario di `command_success` con il campo `Data` a `None` e il campo `logid` contente l'object id del mongolog.

2.2.9 getshells

```
def getshells():

    with open('/etc/shells') as opened:
        shells = opened.read().splitlines()
```

```
    #Removing comment lines
    shells = list( filter( lambda shell: not shell.startswith('#'),
        shells) )

    #Manually Appending dummy shells
    shells = shells + ['/usr/sbin/nologin', '/bin/false']

    return command_success( data=shells )
```

Alcuni delle funzioni del modulo di nomodo `user` richiedono che gli sia passata il percorso di una shell come parametro. Per portare al minimo gli errori si è creata questa funzione che non fa altro che restituire la lista delle shell installate nel sistema, in modo che l'utente non debba immettere manualmente la shell ma deve selezionarla da una lista. Un esempio è quando si chiama la funzione `updateusershell`. Evitiamo così sia errori volontari che errori di scrittura.

Parametri La funzione nella sua semplicità non prende parametri.

Funzionamento

- Apre il file che contiene le shell `/etc/shells` in lettura, ne legge il contenuto, lo divide per linee creando una lista e inserisce tale lista nella variabile `shells`
- In una funzione `filter` legge le righe una alla volta e se una riga inizia per # (cancelletto) la rimuove dalla lista. La verifica è effettuata tramite la funzione `startswith` che restituisce `True` se una stringa inizia col carattere passato come argomento
- Nella terza parte del codice allla lista delle shells aggiunge le shell dummy utilizzate per impedire l'accesso con l'utente che ha quella shell assegnata. Queste shell vengono usate per utenti di servizio, come l'utente `www-data`
- Restituisce le shell nella solita funzione di successo `command_success`

Return Restituisce una lista dove ogni elemento è il percorso di una delle shell installate nel sistema.

2.2.10 updateusershell

```
def updateusershell(user, shell):

    logid = mongolog( locals() )

    if not shell:
        return command_error( returncode=200, stderr="La stringa
            contenente il nome della shell non puo' essere vuota")


    command = ['chsh', user, '-s', shell]

    try:
        check_call(command)
    except CalledProcessError as e:
```

```
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

La funzione nasce per aggiornare la shell assegnata ad un utente.

Parametri

- `user`: è lo username dell'utente a cui verrà cambiata la shell
- `shell`: è la nuova shell da assegnare all'utente. La lista delle shell che si possono utilizzare può essere ricavata dalla funzione getshells

Funzionamento

```
    logid = mongolog( locals() )

    if not shell:
        return command_error( returncode=200, stderr="La stringa
            contenente il nome della shell non puo' essere vuota")
```

L'operazione risulta *sensibile*, viene quindi prima di tutto memorizzato un mongolog con le informazioni sull'operazione.
Viene poi controllato che il parametro `shell` non sia una stringa vuota, per evitare errori con la bash.

```
    command = ['chsh', user, '-s', shell]

    try:
        check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

Viene poi lanciato il comando `chsh` specificando nome utente e shell usando `check_call` e verificando che non sia generata una eccezione. Dopo il cambio delle shell se l'operazione è andata a buon fine quindi si chiama la funzione di successo `command_success` passandogli l'object id del mongolog.

Return Restituisce il dizionario di successo con la variabile `Data` a `None` e `logid` contenente l'object id del log su mongo contenente le informazioni sull'operazione appena effettuata.

2.2.11 adduser

```
def adduser(user, password, shell="/bin/bash"):

    logid = mongolog( locals() )

    if not shell:
        return command_error( returncode=200, stderr="La stringa
            contenente il nome della shell non puo' essere vuota" )
```

```
    try:
        command = ['useradd', '-m', '-p', password, '-s', shell, user]
        check_output(command, stderr=PIPE, universal_newlines=True)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

Serve ad aggiungere nuovi utenti al sistema.

Parametri

- `user`: è il nome utente del nuovo utente che si sta creando. Deve essere univoco nel sistema
- `password`: la password di accesso del nuovo utente
- `shell="/bin/bash"`: è la shell che verrà assegnata all'utente all'atto di creazione. Se questo parametro non viene passato viene assegnata la shell di default di sistema che è `bash`

Funzionamento

```
    logid = mongolog( locals() )

    if not shell:
        return command_error( returncode=200, stderr="La stringa
            contenente il nome della shell non puo' essere vuota" )
```

Viene creato un mongolog in quanto operazione sensibile che va a modificare il sistema e potrebbe compromettere la sicurezza dello stesso. Come in 2.2.10 viene verificato che il parametro `shell` non sia una stringa vuota per evitare errori con la bash.

```
    try:
        command = ['useradd', '-m', '-p', password, '-s', shell, user]
        check_output(command, stderr=PIPE, universal_newlines=True)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

Viene poi composto il comando `useradd` definendo tutte le specifiche dell'utente ed indicando di creare una home col parametro `-m`, viene eseguito e viene controllato che non venga lanciata nessuna eccezione.
Se tutto va a buon fine viene invocata la funzione di successo `command_success` passandogli l'object id del documento mongo creato contenente le informazioni sull'operazione di creazione dell'utente.

Return Restituisce il dizionario di successo generato da `command_success` con la chiave `Data` a `None` e la chiave `logid` contenente l'object id del documento creato e contenente le informazioni sull'operazione eseguita.

2.2.12 removeuser

```
def removeuser(user, removehome=False):

    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )


    try:
        command = ['deluser', user]
        if removehome: command.append('--remove-home')

        check_output( command, stderr=PIPE, universal_newlines=True )
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

È la funzione opposta a adduser e serve a rimuovere un utente dal sistema.

Parametri

- `user`: è il nome utente dell'utente da eliminare dal sistema
- `removehome=False`: se `True` indica di rimuovere anche la home dell'utente oltre all'utente stesso. Di default è `False` in quanto la home degli utenti potrebbero contenere dati importanti che non si vuole perdere

Funzionamento

```
    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )
```

La rimozione di un utente è una operazione molto importante, viene quindi generato un mongolog contenente, oltre ai parametri con cui è chiamata la funzione, anche le informazioni sull'utente che sta per essere rimosso, ricavate invocando la funzione `getuser()`.

```
    try:
        command = ['deluser', user]
        if removehome: command.append('--remove-home')
```

```
        check_output( command, stderr=PIPE, universal_newlines=True )
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

All'atto del lancio del comando `deluser` viene quindi verificato il parametro `removehome`; se questo risulta essere `True` viene aggiunto al comando `deluser <user>` anche il parametro `--remove-home` istruendo quindi lo stesso a rimuovere anche la cartella `/home/<user>/`.
Viene controllato quindi che non venga generata l'eccezione `CalledProcessError` e se tutto va a buon fine viene invocata la funzione di successo `command_success` passandogli l'object id del documento mongo che tiene traccia delle informazioni dell'operazione.

Return Restituisce il dizionario di successo generato da `command_success` con la key `Data` a `None` e la key `logid` contenente l'object id del documento mongo creato e che tiene traccia delle informazioni sull'operazione.

2.3 Network

La sezione network raccoglie tutte le funzioni utili per la gestione della rete, quali creazione e distruzione di interfacce, cambio indirizzo, settaggio rotte ecc.

2.3.1 ifacestat

```
def ifacestat(iface="", namesonly=False):

    command = ['ifconfig', '-a']
    if iface: command.append(iface)

    try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True)
    except CalledProcessError as e:
        return command_error( e, command )


    output = output.split('\n\n')
    del output[-1]


    ifaces = list() if namesonly else dict()
    for iface in output:

        iface = iface.splitlines()
        firstline = iface.pop(0).split(None, maxsplit=1)


        if namesonly:
```

```
            ifaces.append( firstline[0] )
        else:
            iface.insert(0, firstline[1])

            i = 0
            for index, value in enumerate(iface):
                i += 1
                iface[index] = iface[index].strip()
                if iface[index].startswith('UP') or iface[index].
                    startswith('DOWN'):  break

            ifaces.update({ firstline[0]: iface[:i] })

    return command_success( data=ifaces )
```

La funzione serve ad ottenere uno sommario sullo stato delle interfacce di rete attualmente installate su sistema.

Parametri

- `iface=""`: Se non nullo (come di defaut) questo parametro serve a farsi restituire dalla funzione lo stato della sola interfaccia indicata. Questa variabile deve contenere il nome esatto dell'interfaccia di cui si vogliono le informazioni (es. "ens1")
- `namesonly=False`: se a `True` restituisce i soli nomi dell'interfaccia invece di tutte le informazioni

Funzionamento

```
    command = ['ifconfig', '-a']
    if iface: command.append(iface)

    try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True)
    except CalledProcessError as e:
        return command_error( e, command )
```

Il comando che restituisce l'output più adatto è `ifconfig`, viene quindi lanciato usando `check_call()` e l'output conservato nella variabile `output`. Notare che ad `ifconfig` si è aggiunto il parametro `-a` in quanto vogliamo ottenere le informazioni su tutte le interfacce e non solo sulle interfacce attualemente abilitate. Se il parametro `iface` è non nullo questa stringa viene aggiunta al comando così da ottenere solo le info di questa interfaccia.[5]

```
    output = output.split('\n\n')
    del output[-1]

    ifaces = list() if namesonly else dict()
```

`ifconfig` divide le informazioni sulle interfacce con una linea vuota e la fine dell'output con due linee vuote. Dividiamo quindi l'output splittando per `"\n\n"` ottenendo una lista di N+1 stringhe dove

[5] Anche se `iface` è settato non viene comunque rimosso il parametro `-a` in quanto non interferisce con la buona esecuzione del comando

N è il numero delle interfacce del sistema e 1 è la stringa vuota che eliminiamo lanciando il secondo comando (`del`).
Viene poi creata la variabile `ifaces` che conterrà le informazioni da restituire all'utente. Se il parametro `namesonly` è settato a `True` viene inizializzata come lista (una lista di nomi), come dizionario altrimenti.

```
    for iface in output:

        iface = iface.splitlines()
        firstline = iface.pop(0).split(None, maxsplit=1)

        if namesonly:
            ifaces.append( firstline[0] )
        else:
            iface.insert(0, firstline[1])
            i = 0
            for index, value in enumerate(iface):
                i += 1
                iface[index] = iface[index].strip()
                if iface[index].startswith('UP') or iface[index].
                   startswith('DOWN'):  break

        ifaces.update({ firstline[0]: iface[:i] })

    return command_success( data=ifaces )
```

Andiamo quindi ad iterare su queste interfacce che abbiamo ricavato. Per ogni intrefaccia ricavata:

- Si dividono le informazioni sulle interfacce per riga (splitlines())
- La prima riga contiene il nome dell'interfaccia più alcune informazioni divisi da uno spazio (il primo spazio della riga); si va quindi a rimovere questa riga (pop(0)), e la si divide per il primo spazio ottenendo `firstline` che è una lista dove il primo elemento è il nome dell'interfaccia mentre il secondo è la prima riga di informazioni
- Se `namesonly` è `True` si vogliono solo i nomi delle interfacce e si inserisce in `ifaces` (usata come lista) il nome dell'interfaccia contenuta in `firstline[0]` per poi passare alla prossima iterazione
- Se invece `namesonly` è `False`, `ifaces` è un dizionario. Questo dizionario verrà creato in modo che la chiave sia il nome dell'interfaccia mentre il valore siano le sue informazioni. Andiamo quindi prima di tutto ad inserire la prima riga di informazioni (contenuta in `firstline[1]` nella variabile `iface` in modo che questa variabile contenga nuovamente la lista completa delle informazioni sull'interfaccia
- L'output di `ifconfig` porta con se delle informazioni che attualmente a noi non servono. Per eliminare queste informazioni che risiedono nelle ultime righe della variabile `iface` si esesegue questo secondo ciclo for. Il ciclo viene eseguito usando `enumerate` in quanto abbiamo bisogno di modificare la variabile `iface` originale e non una sua copia, per motivi in seguito spiegati.
  Si costituisce un indice `i` che indica la riga su cui stiamo iterando e:
  - Viene incrementato l'indice che identifica la riga sulla quale si sta iterando
  - Vengono rimossi gli spazi iniziali e finali della riga. Da qui l'uso di `enumerate`

  - Se la riga inizia per *UP* oppure *DOWN* allora abbiamo raggiunto l'ultima riga che contiene informazioni a noi utili. Si lancia quindi un break uscendo dal ciclo; la varibile `i` a questo punto contiene l'esatta riga dove fermarsi e potrà essere sfruttata per tagliare le righe che non ci servono

- Usciti dal ciclo non ci resta che tagliare le righe che non ci servono ed inserire nel dizionario `ifaces` la chiave e il valore. Viene quindi lanciato `ifacea.update()` dove si passa come chiave `firstline[0]` che è il nome dell'interfaccia e come valore `iface[:i]` che solo le informazioni sulla stessa tagliate fino alla riga `i`

Alla fine del parsing, come solito per le funzioni di nomodo, viene chiamata la funzione di successo `command_success` passandogli nel parametro `data` il dizionario appena creato `ifaces`.

Return

- Se `namesonly` è `True` restituisce una lista di stringhe dove ogni stringa è il nome di una intrefaccia presente nel sistema
- Se `namesonly` è `False` restituisce un dizionario dove la chiave è il nome delle interfacce di sistema mentre il valore sono le informazioni sulla stessa, come ad esempio indirizzo IPv4 e IPv6, MAC address, stato ecc.

2.3.2 getnewifacealiasname

```
def getnewifacealiasname(iface):

    ifaces = ifacestat( namesonly=True )
    if ifaces['returncode'] is 0:
        ifaces = ifaces['data']


    aliasid = 0
    for item in ifaces:
        if item.startswith( iface + ':' ):
            item = int(re.sub('.*:', '', item))
            if aliasid is item: aliasid += 1
            else: break

    return command_success( data = iface + ':' + str(aliasid) )
```

Questa funzione non fa altro che restituire al chiamante il possibile nome di una nuova interfaccia alias data un'interfaccia fisica esistente. È intesa per essere usata come titolo della pagina in cui l'utente andrà ad inserire le specifiche della nuova interfaccia e per essere passata alla funzione createalias per la creazione dell'interfaccia virtuale.

Parametri Prende un solo parametro `iface` che è l'interfaccia a cui fare riferimento per la creazione del nuovo alias.

Funzionamento La funzione si basa su queste 3 considerazioni:

1. Il nome di una interfaccia alias è costitita dal nome dell'interfaccia più un due punti più un identificativo numerico per l'alias stesso che parta da 0

2. Data una interfaccia, ad es. `ens1`, se per questa interfaccia sono già presenti degli alias, ad es. `ens1:1` e `ens1:2`, deve essere restituito all'utente il nome alias `ens1:3`

3. Per evitare un numero sempre crescente per i nomi alias un nuovo nome dovrà riempire il buco lasciato da un interfaccia che sia stata eliminata. Ad es. se abbiamo `ens1:0` e `ens1:2` la nuova interfaccia dovrà avere il nome `ens1:1`

Si discutono i metodi utilizzati per soddisfare tali principi.

```
    ifaces = ifacestat( namesonly=True )
    if ifaces['returncode'] is 0:
        ifaces = ifaces['data']
```

Viene chiamata la funzione ifacestat con il parametro `namesonly` impostato a `True` così da ottenere i nomi di tutte le interfacce del sistema; viene inoltre fatto un controllo sul codice di ritorno della funzione.

```
    aliasid = 0
    for item in ifaces:
        if item.startswith( iface + ':' ):
            item = int(re.sub('.*:', '', item))
            if aliasid is item: aliasid += 1
            else: break

    return command_success( data = iface + ':' + str(aliasid) )
```

Si inizializza quindi una variabile `aliasid` al primo numero possibile per gli identificativi alias, cioè 0 (ad esempio `ens1:0`).
Per capire quanto a seguire bisogna tenere in mente che l'output di ifconfig ordina gli alias di un'interfaccia; ad esempio con `ens1` l'ordine sarà `ens1, ens1:0, ens1:2` e mai ad esempio `ens1, ens1:2, ens1:0`
Con tale considerazione si inizia iterando sui nomi delle interfacce. Se questa è un alias dell'interfaccia passata come parametro, cioè se il suo nome inizia (`startswith`) con il nome dell'interfaccia `iface` seguita dal due punti si entra nell'if che si vede in codice. In questo `if` si elimina il nome dell'interfaccia ed i due punti utilizzand la libreria `re` (ad esempio `ens0:1` diventa semplicemente 1), ottenendo il numero identificativo della prima interfaccia alias; si confronta quindi questo numero con `aliasid`. Se questi sono uguali l'interfaccia alias con identificativo `aliasid` è sicuramente occupata, quindi si itera e si tenta nuovamente con il prossimo alias; se invece `aliasid` non coincide col numero identificativo del primo alias allora il posto è libero e si esce dal ciclo con un break.
Dopodichè basta chiamare la funzione di successo `command_success` passandogli nel campo `data` il nome della nuova interfaccia, che non è altro che il nome della stessa seguita dal 2 punti e dal numero identificativo dell'alias contenuto nella variabile `aliasid`.

Return Restituisce il nome del nuovo alias dell'interfaccia `iface` da utilizzare come titolo della pagina in cui l'utente andrà ad inserire le specifiche della nuova interfaccia o per essere passata alla funzione createalias per la creazione dell'interfaccia virtuale.

2.3.3 ifacedown

```
def ifacedown( iface ):

    logid = mongolog( locals() )

    command = ['ifconfig', iface, 'down']

    try:
        check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )
```

La funzione è intesa per disattivare un'interfaccia di rete.

Parametri Prender un solo parametro che è l'interfaccia di rete da disattivare.

Funzionamento Essendo un'operazione sensibile prima di tutto crea un mongolog e ne memorizza l'object id nella variabile `logid`.
Poi semplicemente costruisce il comando in questo modo: `ifconfig <nome_interfaccia> down` e lo lancia usando `check_call` (in quanto non è restituiuto output) e verifica che non siano avvenute eccezioni. Chiama all fine la funzione di successo `command_success` passandoli nel campo `logid` l'object id del documento mongo creato e contenenti le specifiche dell'operazione appena eseguita.

Return Restituisce il dizionario di successo con il campo `data` a `None` e il campo `logid` contenente l'object id del documento mongo che contiene le specifiche dell'operazione eseguita.

2.3.4 ifaceup

```
def ifaceup( iface, address="", netmask="", broadcast="" ):

    logid = mongolog( locals() )

    command = ['ifconfig', iface]
    if address: command.append(address)
    if netmask: command = command + ['netmask', netmask]
    if broadcast: command = command + ['broadcast', broadcast]
    command.append('up')

    try:
        check_output = check_call(command, stderr=PIPE,
            universal_newlines=True)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )
```

È l'opposto della funzione ifacedown e serve ad abilitare un'intrefaccia di rete attualmente disabilitata.

Parametri Prende 4 parametri:

- `iface`: il nome dell'interfaccia da tirare sù
- `address=""`: è l'indirizzo da assegnare all'interfaccia. Non è obbligatorio in quanto l'interfaccia potrebbe già avere un indirizzo assegnato
- `netmask=""`: è la maschera di rete da assegnare all'interfaccia. Non è obbligatorio in quanto l'interfaccia potrebbe già avere una maschera di rete settata
- `broadcast=""`: è l'indirizzo facente parte della stessa rete dell'interfaccia, sulla quale i pacchetti vengono inviati in broadcast a tutti gli host della rete stessa. Non è obbligatorio in quanto l'interfaccia potrebbe già avere un indirizzo broadcast settata

Funzionamento Si memorizza prima di tutto un mongolog per tenere traccia dell'operazione.
Durante la fase di costruzione del comando semplicemente di controlla ad uno ad uno se è stato assegnato un valore ai parametri; in tale caso si appende al comando da lanciare. Uno volta controllati tutti i parametri si appende la parola `up` (che indica che l'interfaccia va abilitata) e si lancia il comando in una `check_call` (nessun output generato) controllando che non si sia verificata un'eccezione.
Se tutto è andato a buon fine si chiama la funzione di successo `command_success` passandogli l'object id del documento mongo creato.

Return Il dizionario di successo della funzione `command_success` col campo `data` a `None` e il campo `logid` contenente l'object id del documento mongo creato e contenente le specifiche dell'operazione appena effettuata.

2.3.5 createalias

```
def createalias( aliasname, address, netmask="", broadcast="" ):

    iface = re.sub(':.*', '', aliasname)

    aliases = ifacestat( namesonly=True )
    if aliases['returncode'] is 0:
        if not iface in aliases['data']:
            return command_error( returncode=197, stderr="No interface
                found with such name: " + iface )
    else:
        return aliases

    return ifaceup( iface=aliasname, address=address, netmask=netmask,
        broadcast=broadcast )
```

Nasce per consentire la creazione di una interfaccia alias dal pannello web di nomodo. Utilizza la funzione ifaceup.

Parametri Accetta gli stessi parametri di `ifaceup`, che sono i seguenti:

- `aliasname`: È il nome della nuova interfaccia alias da creare. Può essere ricavata usando la funzione getnewifacealiasname
- `address`: L'indirizzo della nuova interfaccia. È l'unico parametro obbligatorio insieme al nome dell'alias
- `netmask=""`: La maschera di rete per la nuova interfaccia. Non è obbligatorio e se non passato viene calcolata dal sistema
- `broadcast=""`: L'indirizzo di broadcast su cui l'interfaccia invia i pacchetti destinati a tutti gli host della rete. Non è obbligatorio e se non passato viene calcolato dal sistema

Funzionamento Il nome dell'interfaccia da creare può essere ottenuto chiamando la funzione getnewifacealiasname ma è comunque data la possibilità all'utente di scegliere un nome personalizzato. A causa di ciò a differenza delle altre funzioni in `createalias` bisogna verificare almeno se l'interfaccia di cui si vuole creare l'alias esiste.
Viene quindi estratta dal parametro `aliasname` il nome dell'interfaccia principale, eliminando il due punti e l'identificativo dell'alias, usando la libreria per le espressioni regolari `re`. Ad esempio se `aliasname` è `ens1:0` si elimina `:0` ottenendo `ens1`.
Dopodichè si chiama la funzione ifacestat col parametro `namesonly` settato a `True` per ottenere i nomi di tutte le interfacce del sistema.
Viene ovviamente verificato che le operazioni della funzione chiamata non abbiano generato errori verificare la chiave `returncode` del dizionario ottenuto da tale funzione; se `returncode` è diverso da 0 viene restituito il dizionario di errore così come è stato ottenuto al chiamante. Se invece l'operazione è andata a buon fine si verifica che l'interfaccia ottenuta dal precedente `re.sub` sia presente nella lista di interfacce del sistema. Se non presenta viene generato un errore personalizzato chiamando la funzione `command_error`, mentre se presente semplicemente viene chiamata la funzione precedentemente discussa ifaceup che passandogli paripari i parametri inseriti dalll'utente. Con la sua sintassi questa funzione riesce anche a creare una interfaccia alias.

Return Restistuisce il dizionario di ritorno generato dalla funzione ifaceup.

2.3.6 destroyalias

```
def destroyalias( aliasname ):
    return ifacedown( aliasname )
```

È l'inverso della funzione createalias e serve per rimuovere una interfaccia alias dal sistema.

Parametri Accetta un solo parametro `aliasname` che è il nome dell'interfaccia alias da distruggere.

Funzionamento A differenza della funzione createalias qui non c'è bisogno di verificare se l'interfaccia esista in quanto all'utente non è data la possibilità di definire una interfaccia alias personalizzata da distruggere ma semmplicemente ne sceglie una da una lista di intrefacce presenti nel sistema.
Viene quindi semplicemente chiamata la funzione ifacedown passandogli il nome dell'interfaccia.

Return Restituisce il dizionario di ritorno generato dalla funzione ifacedown.

2.3.7 editiface

```
def editiface( iface, address="", netmask="", broadcast="" ):
    return ifaceup( iface=iface, address=address, netmask=netmask,
        broadcast=broadcast )
```

La funzione consente di modificare dei parametri dell'interfaccia, quali indirizzo, maschera di rete e indirizzo di broadcast.

Parametri Prende fino a 4 parametri:

- `iface`: è il nome dell'interfaccia da modificare
- `address=""`: non è obbligatorio e se passato indica il nuovo indirizzo che l'interfaccia `iface` deve avere
- `netmask=""`: non è obbligatorio e se passato indica la nuova maschera di rete che l'interfaccia `iface` deve avere
- `broadcast=""`: non è obbligatorio e se passato indica il nuovo indirizzo di broadcast che l'interfaccia `iface` deve avere

Funzionamento Non è obbligatorio verificare se l'interfaccia `iface` è presente nel sistema (così come fa createalias) in quanto l'utente non ha la possibilità di specificare a mano l'interfaccia da modificare ma semplicemente la sceglie da una lista.
Semplicemenete chiama la funzione ifaceup passandogli tutti i parametri che ha ricevuto; a causa della sua sintassi generica questa funzione consente anche la modifica di interfacce.

Return Restituisce il dizionario di ritorno generato dalla funzione ifaceup.

2.3.8 getroutes

```
def getroutes():

    command = ['route', '-n']

    try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )


    output.pop(0)
    header = output.pop(0).split()
    routes = list( map( lambda route: dict(zip(header, route.split())),
        output ) )


    return command_success( data=routes )
```

La funzione serve ad ottenere la lista di tutte le rotte utilizzate dal sistema per uscire sulla rete.

Parametri Non prende parametri.

Funzionamento

```
    command = ['route', '-n']

    try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )
```

Nella prima parte del codice viene costruito il comando da lanciare per ottenere le rotte. Si è usato il comando `route` passandogli il parametro `-n` in modo da ottenere la lista delle rotte senza che gli indirizzi di rete vengano risolti.
Una voilta costruito il comando viene lanciato usando la funzione `check_output` che oltre ad eseguirlo restituisce anche l'output dello stesso. In caso venga generata l'eccezione `CalledProcessError` questa viene catturata e passata al comando command_error che crea un dizionario di errore che viene restituito all'utente.

```
    output.pop(0)
    header = output.pop(0).split()
    routes = list( map( lambda route: dict(zip(header, route.split())),
        output ) )

    return command_success( data=routes )
```

Nella seconda parte prima di tutto viene eliminato la prima riga del comando che solitamente contiene la stringa *Kernel IP routing table* (a noi non utile) usando la funzione `pop()`.
Viene poi rimosso dall'output e splittato l'header; questo header contiene la descrizione breve dei campi della tabella di routing, e solitamente è il seguente:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use
    Iface
```

Nell'ultima parte usando la funzione `map()` viene creata una lista di dizionari in cui:

- Ogni elemento delle lista contiene un dizionario con le specifiche di una rotta
- Ogni dizionario ha tanti campi quanti ne ha l'header, e ogni campo contine come chiave un elemento dell'header e come valore il valore della rotta inerente a quell'elemento header

Un esempio vale più di mille parole, quindi ecco un esempio del dizionario generato su un sistema Ubuntu Server 16.04 che gira su un container LXD:

```
[{'Destination': '0.0.0.0',
  'Flags': 'UG',
  'Gateway': '10.100.10.1',
  'Genmask': '0.0.0.0',
  'Iface': 'eth0',
  'Metric': '0',
  'Ref': '0',
  'Use': '0'},
{'Destination': '10.0.0.0',
```

```
  'Flags': 'U',
  'Gateway': '0.0.0.0',
  'Genmask': '255.0.0.0',
  'Iface': 'eth0',
  'Metric': '0',
  'Ref': '0',
  'Use': '0'},
{'Destination': '10.100.10.0',
  'Flags': 'U',
  'Gateway': '0.0.0.0',
  'Genmask': '255.255.255.0',
  'Iface': 'eth0',
  'Metric': '0',
  'Ref': '0',
  'Use': '0'}]
```

Alla fine viene chiamata la funzione di successo `command_success` passandogli il dizionario delle rotte nel campo data.

Return Restituisce il dizionario di successo, con il campo `data` contenente il dizionario delle rotte descritto nel paragrafo **funzionamento**.

2.3.9 addroute

```
def addroute(gw, net, netmask, default=False):

    logid = mongolog( locals() )

    command = ['route', 'add']
    if default:
        command = command + ['default', 'gw', gw]
    elif net is None or netmask is None:
        command_error( returncode=201, stderr='On non-default route you
            must enter "net" and "netmask" parameters' )
    else:
        command = command + ['-net', net, 'netmask', netmask, 'gw', gw]

    try:
        check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )
```

La funzione nasce per aggiungere una rotta alle rotte già presenti nel sistema, visualizzabili chiamando la funzione getroutes.

Parametri Accetta 4 parametri di cui la maggior parte obbligatori:

- `gw`: è l'indirizzo su cui uscire sulla rete `net`

- `net`: rappresenta la rete a di destinazione a cui la rotta fa riferimento
- `netmask`: è la maschera di rete della rete di destinazione, identificata dal parametro `net`
- `default=False`: se `True` indica di creare un default gateway

Funzionamento La maggior parte della funzione si basa sulla costruzione del comando. Le prime due keyword sono predenfinite e sono `route add` che indicano la creazione di una nuova rotta. Entrando nell'`if` distinguiamo tre situazioni:

- Il parametro `default` è `True`: si vuole creare un default gateway, si ha quindi solo bisogno del gateway identificato dal parametro `gw`. Il comando viene costruito inserendo questo unico parametro
- `default` è `False` ma i parametri sono `None` o stringhe vuote: c'è stato un errore nella chiamata, se `default` è `False` per costruire la rotta c'è bisogno di tutti e tre i parametri `gw`, `net` e `netmask`
- `default` è `False` e si hanno tutti i parametri: si vuole creare una nuova rotta, e il comando viene costruito inserendo tutti questi parametri

Dopo la costruzione del comando lo si lancia usando la `check_call` (in quanto non viene restituito output) e si chiama la funzione di errore o di successo a secondo del risultato. Alla funzione di successo viene passato l'object id del documento mongo creato.

Return Restituisce il dizionario di successo con la chiave `data` a `None` (nessun output restituito) e la chiave `logid` contenente l'object id del mongolog creato contenente le informazioni sull'operazione eseguita.

2.3.10 defaultroute

```
def defaultroute(gw): return addroute(gw, net=None, netmask=None, default
    =True)
```

La funzione è definita come *Funzione alias* in quanto basata quasi completamente su un'altar funzione. Prende in carico di creare un default gateway chiamando la funzione addroute con i giusti parametri.

Parametri Prende un solo parametro che è il gateway da assegnare.

Funzionamento Semplicemente chiamta la funzione addroute passandogli il gateway nel parametro `gw`, settando la variabile `default` a `True` e le altre variabili a `None`.

Return Restituisce il dizionario di ritorno della funzione addroute.

2.3.11 delroute

```
def delroute(route):

    logid = mongolog( locals() )

    if not type(route) is type(dict()):
```

```
        command_error( returncode=202, stderr='delroute function can only
            accept a dictionary as argument' )

    command = [ 'route', 'del', '-net', route['Destination'], 'netmask',
        route['Genmask'], 'gw', route['Gateway'] ]

    try:
        check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )
```

La funzione nasce per cancellare una delle rotte presenti nel sistema.
ATTENZIONE: leggere attentamente il paragrafo *Prametri* prima di utilizzarla, in quanto è l'unica che si comporta direttamente sulla questione "parametri in ingresso".

Parametri Accetta un solo parametro che è un dizionario contenente le specifiche della rotta da eliminare. Questo dizionario deve essere lo stesso che si ottiene chiamando la funzione getroutes. Ad esempio prendendo come riferimento l'esempio in sezione 2.3.8 al paragrafo *Funzionamento* volendo cancellare l'ultima rotta dovrei prendere il seguente dizionario (che è il terzo) e passarlo per intero al questa funzione:

```
{'Destination': '10.100.10.0',
  'Flags': 'U',
  'Gateway': '0.0.0.0',
  'Genmask': '255.255.255.0',
  'Iface': 'eth0',
  'Metric': '0',
  'Ref': '0',
  'Use': '0'}]
```

Funzionamento L'opreazione è classificata "sensibile" viene quindi creato un mongolog contenente le informazioni sull'operazione. Viene poi controllato il tipo di parametro in ingresso, e se non è un dizionario viene generato un errore personalizzato usando la funzione **??**.
La costruzione del comando avviene come si vede in codice, mixando le direttive alle informazioni contenute nel dizionario in input. Infine viene lanciato il comando, verificato che non sono successe eccezioni e se tutto è andato a buon fine viene chiamata la funzione di successo passandogli l'object id del documento mongo creato.

Return Restituisce il dizionario di successo senza dati (`data=None`) ma con l'object id del mongolog creato alla chiave `logid`.

2.4 Cron

Il cron è un software incluso in tutte le distribuzioni Linux che copre il compito di eseguire detereminati task (definiti dall'utente) con una certa frequenza e a specifici giorni e orari.
Un esempio di task ricorrente può essere ad esempio l'aggiornamento del database dei file per la ricerca indicizzata.

Il cron di Linux può essere di sistema o dell'utente. Siccome usando il cron di sistema è possibile lanciare i processi utilizzando qualsiasi utente del sistema si è deciso di tralasciare il cron utente ed implementare in nomodo solo quello di sistema.
Distinguiamo 2 tipi di cron:

- Il cron utilizzato da `cronspath` e dagli script in `cron.d` che sono veri e propri script di cron, contenente cioè tutte le informazioni sulla tempistica di lancio del comando o dello script, l'utente e il comando stesso. Il comando in questo caso deve essere contenuto in una sola riga. In nomodo la creazione di tali script è possibile chiamando la funzione addcron
- Il cron delle cartelle `/etc/cron.daily`, `/etc/cron.hourly`, `/etc/cron.monthly` e `/etc/cron.weekly` che sono invece degli script o dei programmi (solitamente scrcitti in bash) che vengono eseguiti uno alla volta ordinati per nome all'ora e alla data definiti di default nel file `/ect/crontab`. In nomodo la creazione di tali script avviene tramite la chiamata alle funzioni derivate da adddefaultcron, ad es. `adddailycron`

2.4.1 listcrontabs

```
def listcrontabs():

    basedir = '/etc/'
    paths = ['cron.d', 'cron.daily', 'cron.hourly', 'cron.monthly', 'cron
       .weekly']

    cronlist = dict()
    for path in paths:
        flist = os.listdir(basedir + path)
        flist.remove('.placeholder')
        cronlist.update({ path: flist })

    return command_success( data=cronlist )
```

La funzione restituisce la lista di tutti i crontab installati nel sistema.

Parametri Non prende nessun parametro.

Funzionamento Definisce due variabili, `basedir` che è dove sono presenti le cartelle del cron e `paths` che è una lista contenente i nomi delle cartelle del cron. Notare che la lista delle cartelle non contiene il file `crontabs` che si è deciso di non utilizzare in quanto egregiamente sostituito dalla cartella `cron.d`.
Entrando nel ciclo for costruisce il dizionario `cronslist` la cui chiave è il nome della cartella mentre il valore è la lista dei file presenti nella cartella stessa, che sarebbero quindi la lista dei crontab. Da questa lista viene eliminato il file `.placeholder` che è solo un file richiesto dal sistema e non utile all'utente finale.

Return Restitusice il dizionario di successo contenente nella variabile `data` il dizionario creato dalla funzione come spiegato nel funzionamento. Un esempio del dizionario è il seguente:

2.4.2 getcroncontent

```
def getcroncontent(cronpath):

    try:
        with open(cronpath, 'r') as content:
            return command_success( data=content.read() )
    except FileNotFoundError:
        return command_error( returncode=10, stderr='No cron file found: '
           "'+cronpath+'"' )
```

La funzione dato il path di un file di cron restituisce il contenuto dello stesso.

Parametri Accetta l'unico parametro `cronpath` che è il percorso del file di cron di cui leggerne il contenuto.

Funzionamento La funzione semplicemente apre il file in lettura utilizzando `with open()` e ne restituisce il contenuto. Durante le sue operazioni controlla che non venga lanciata l'eccezione `FileNotFoundError` che avviene quando il file che si sta cercando di aprire il lettura non esiste.

Return Restituisce il dizionario di successo contenente alla variabile `data` il contenuto del cron indicato. Restituisce invece il dizionario di errore se viene generata l'eccezione `FileNotFoundError`, con un codice di ritorno e un messaggio di errore personalizzato e che sono quello che si vedono in codice.

2.4.3 getcronname

```
def getcronname(): return 'nomodo-' + datetime.datetime.now().strftime('%
    Y%m%d%H%M%S')
```

Genera un nome da usare per un nuovo cron. Viene usato principalmente in due fasi:

1. Viene chiamato dal frontend per riempire il campo *name* all'aggiunta di un nuovo crontab. L'utente ovviamente ha la possibiltà di cammbiare questo nome prima dell'aggiunta dello script
2. Chiamato dalle funzione di aggiunta cron addcron e adddefaultcron in quanto se il parametro `name` non è stato passato vanno ad utilizzare il nome restituito da questa funzione. Questa operazione sembra ridondante in quanto già chiamata dal frontend come spiegato al punto 1, ma serve a dare solidità al codice

Parametri Non prende nessun parametro.

Funzionamento Semplicemente contatena la stringa "nomodo-" alla data e ora odierna col formato che si vede nel codice.

Return È una delle pochissime funzione che non chiama una delle due funzioni di ritorno di nomodo. Restituisce la stringa costruita come spiegato nel funzionamento.

2.4.4 addcron

```
def addcron( command, name="", user="root", minute='*', hour='*', dom='*'
    , month='*', dow='*' ):

    cronspath = '/etc/cron.d/'

    #New cron gets a random name if user did not provide it
    if not name: name = getcronname()

    logid = mongolog( locals() )

    with open(cronspath + name, 'w') as newcron:
        newcron.write( minute + ' ' + hour + ' ' + dom + ' ' + month + ' '
            ' + dow + ' ' + user + ' ' + command + '\n' )

    return command_success( data=cronspath+name, logid=logid )
```

La funzione nsace per la creazione di uno script di cron vero che si vada ad aggiungere a quelli presenti nella cartella `/etc/cron.d/`. Per "script di cron vero" si intende un file di direttive che contenga la sintassi del cron; come spiegato nell'introduzione alla sezione **??** questi si differenziano dagli script specifici in quanto questi ultimi sono veri e propri applicativi, usualmente scritti in bash, mentre gli script di cron contengono anche le informazioni sulla tempistica dell'esecuzione del comando e il comando deve essere contenuto in una linea o deve essere la chiamata ad un applicativo.
La spiegazione di centos in questo esempio chiarisce ancora meglio come deve essere strutturato uno script di cron:ù

```
# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,
    wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name  command to be executed
```

Si intuisce da questo esempio che mentre i campi `dom` e `dow` interferiscano tra di loro. In realtà settando entrambi il comando verrà eseguito sia nel "day of month" sia nel "day of week" specificari.

Parametri Accetta un sacco di parametri, qui spiegati:

- `command`: È il comando che deve essere eseguito. Deve essere di una sola riga e può essere sia un comando di bash che la chiamata ad un'applicativo di sistema, che può essere in qualsiasi linguaggio, incluso bash stesso (ovviamente)
- `name=""`: È il nome del nuovo script di bash. Se non specificato viene gererato automaticamente attraverso la chiamata alla funzione getcronname. In realtà la funzione viene chiamata direttamente dal frontend per settare il nome, ma per la massima sicurezza viene fatto un controllo per vedere se il nome è vuoto, e generarlo in tal caso

- `user="root"`: È l'utente per conto di cui il comando verrà lanciato. Di default è l'utenza amministrativa.
- `minute="*"`: È il minuto in cui il comando verrà eseguito. Di default è * che indica ogni minuto
- `hour="*"`: È l'ora in cui il comando verrà eseguito. Di default è * che indica ogni ora. Accetta valori da 0 a 24 ed è basato sul fuso orario del sistema
- `dom="*"`: È il giorno del mese (dom="day of month") in cui il comando verrà eseguito. Di default è * che indica ogni giorno del mese. Accetta valori da 1 a 31 se il mese ha 31 giorni
- `month="*"`: È il mese in cui il comamndo deve essere eseguito. Di default è * che indica ogni mese. Accetta valori da 1 da 12
- `dow="*"`: È il giorno della settimana in cui il comando deve essere esguito. Di default è * che indica ogni gionro della settimana. Accetta valori da 1 a 7

Funzionamento Una volta settato il path in cui il file verrà creato (`/etc/cron.d/`) si crea il mongolog in quanto bisogna tenere traccia di chi ha creato il file, e semplicemente usando una `with open` si apre il file in scrittura e si scrivono tutti le informazioni ricevute come parametri.

Return Restituisce il percorso del file creato e il logid del documento mongo contenente le informazioni sulla operazione nel dizionario di successo generato dalla funzione `command_success`.

2.4.5 adddefaultcron

```
def adddefaultcron(command, cronspath, name):

    #New cron gets a random name if user did not provide it
    if not name: name=getcronname()

    logid = mongolog( locals() )

    with open(cronspath + name, 'w') as newcron:
        newcron.write( command + '\n' )

    return command_success( data=cronspath+name, logid=logid )


def addhourlycron(command, name=""): return addefaultcron( name=name,
   command=command, cronspath='/etc/cron.hourly/' )
def adddailycron(command, name=""): return addefaultcron( command=command
   , cronspath='/etc/cron.daily/' )
def addweeklycron(command, name=""): return addefaultcron( command=
   command, cronspath='/etc/cron.weekly/' )
def addmonthlyycron(command, name=""): return addefaultcron( command=
   command, cronspath='/etc/cron.monthly/' )
```

La funzione si contrappone a addcron e a differenza di questo serve a creare uno script nelle cartelle `/etc/cron.hourly`, `/etc/cron.daily`, `/etc/cron.weekly` e `cron.monthly`. Come spiegato gli script

di queste cartelle a differenza di quelli in `/etc/cron.d` sono veri e propri script (solitamente script di bash) che vengono eseguiti ordinati per nome e secondo le direttive del file `/etc/crontab`. Qui un esempio di tale file preso da un Ubuntu 16.04 server:

```
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the 'crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user  command
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts
   --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts
   --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts
   --report /etc/cron.monthly )
```

Quindi ad esempio tutti gli script in `/etc/cron.daily` verranno eseguiti ogni mattina alle ore 6:25.

Parametri Non prendendo informazioni ne sulla tempistica ne sull'utente prende pochi parametri, qui presentati:

- `command`: È il contenuto dello script che verrà aggiunto alle cartelle. In questo caso può essere multilinea. Il fontend può leggere questo contenuto ad esempio da una text area
- `cronspath`: Dato che questa funzione non andrebbe mami richiamata direttamente ma sempre attraverso le 4 funzioni che si vedono nelle ultime 4 righe del codice questo è un parametro che indica la cartella dove lo script sarà posizionato; questo cambia a seconda della funzione chiamata
- `name`: È il nome dello script di cron, che come per addcron se non presente viene generato tramite la chiamata alla funzione getcronname

Funzionamento Come per `addcron` la funzione non fa altro che verificare che il nome sia presente e generarlo in caso negativo, creare un documento mongo che memorizzi l'operazione effettuata e scrivere lo script (contenuto nella variabile `command`) nel nuovo file.
La funzione principale non va mai chiamata direttamente ma bisogna sempre utilizzare le 4 funzioni intermediare che si vedono nelle ultime 4 righe del codice e che vanno a settare correttamente la variabile `cronspath` a seconda della tempistica di esecuzione dello script.

Return Ugualemente a `addcron` restituisce il dizionario di successo contenete il percorso del nuovo file nella variabile `data` e il l'object id del documento mongo contenente le informazioni sull'operazione nella variabile `logid`.

2.4.6 writecron

```
def writecron( cronpath, newcontent ):
    return writefile( filepath=cronpath, newcontent=newcontent+'\n' )
```

Nasce per scrivere il contenuto di un file di cron. Ovviamente può essere usato per sovrascrivere completamente il contenuto o in congiunzione con getcroncontent per aggiornarne il contenuto.

Parametri Prende 2 parametri:

- `cronpath`: È il percorso del file di cron che verrà scritto
- `newcontent`: È il nuovo contenuto del file che verrà scritto nel file indicato in `cronpath`

Funzionamento La funzione è basata su writefile spiegata precedentemente.

Return Restituisce ciò che ritorna writefile.

2.4.7 removecron

```
def removecron(cronpath): delfile( path=cronpath )
```

Serve ad eliminare un file di cron e quindi a fermare l'esecuzione del comando in esso contenuto.

Parametri Accetta un solo parametro che è il percorso del file da eliminare.

Funzionamento La funzione è basata sulla funzione precedentemente spiegata delfile.

Return Restituisce ciò che ritora la funzione delfile.

2.5 Sistema

Questa libreria è relativamente piccola per adesso ma sarà riempita con altre funzione in seguito. Contiene tutte quelle funzione che riguardano il sistema in generale, come ad esempio la funzione per la visualizzione e il cambio del nome macchina.

2.5.1 hostname

```
def hostname(newhostname=""):

    command = ['hostname']

    if newhostname:
        logid = mongolog( locals() )
        command.append(newhostname)

    try:
        hostname = check_output(command, stderr=PIPE, universal_newlines=
            True)
```

```
    except CalledProcessError as e:
        return command_error( e, command )

    if newhostname:
        return command_success( logid=logid )
    else:
        return command_success( data=hostname )
```

La funzione serve sia ad ottenere il nome macchina che a cambiarlo.

Parametri Accetta un solo parametro opzionale `newhostname` che contiene il nuovo nome macchina da assegnare alla stessa.

Funzionamento Si comporta similmente alla funzione `hostname` dei sistemi linux, ossia se chiamata senza alcun parametro restituisce l'hostname della macchina sulla quale viene eseguito, mentre e chiamato con un parametro allora setta il contenuto di questo parametro come nuovo nome macchina. Viene quindi fatto un controllo, se il parametro non obbligatorio `newhostname` contiene una stringa allora crea un nuovo momgolog e assegna alla macchina questa stringa come nome host, restituisce l'hostname della macchina altrimenti.

Return Restituisce il dizionario di successo contenente l'hostname se `newhostname` è vuota, l'object id del mongolog altrimenti.

2.5.2 getsysteminfo

```
def getsysteminfo( getall=True, getproc=False, getcpu=False, getmem=False
    ):

    #Tuple to return
    toreturn = ()

    ##### CPU #####
    if getall or getcpu:

        #Reading cpu stat from /opt files
        with open('/proc/cpuinfo', 'r') as cpuorig:
            cpuraw = cpuorig.read().splitlines()

        #Removing duplicate lines by converting list() to set()
        cpuraw = set(cpuraw)

        #Removing empty lines
        cpuraw = list( filter( None, cpuraw ) )

        #Removing useless lines
        linestoremove = ('flags', 'apicid', 'processor', 'core_id', '
            coreid')
        cpuraw = list( filter( lambda line: not any(s in line for s in
            linestoremove), cpuraw ) )
```

```
        #Deleting all tabulation and spaces for each line of the cpuraw
            set cpuraw
        cpuraw = map( lambda line: re.sub('[\t| ]*:[\t| ]*', ':', line),
            cpuraw )


        #We got three fields named "cpu Mhz", but to use them as
            dictionry keys
        #we need to rename them all
        cpuaf = list()
        i = 1
        for line in cpuraw:
            #Adds an incremental number to the key
            if 'mhz' in line.lower():
                cpuaf.append( re.sub('^.*:', 'core' + str(i) +' MHz:',
                    line) )
                i += 1
            else: cpuaf.append( line )


        #Buiding final dictionary cotaining cpu information in the right
            form
        cpu = dict()
        for line in cpuaf:
            line = line.split(':')
            cpu.update({ line[0]: line[1] })

        #Adding cpu dict to the tuple to return
        toreturn = toreturn + (cpu,)



    ##### MEMORY #####
    if getall or getmem:

        #Reading memory status from /opt files
        with open('/proc/meminfo', 'r') as memorig:
            memraw = memorig.read().splitlines()


        #Filling mem dict with memory information
        mem = dict()
        for line in memraw:
            line = re.sub(' ', '', line)                    #Removing spaces
                for each line
            line = line.split(':')                          #Splitting by
                colon
```

```
            mem.update({ line[0].lower() : line[1] })   #Appending the
                dictionary to a list to return

        toreturn = toreturn + (mem,)




    #### PROCESSES #####
    if getall or getproc:

        #Reading processes status using top command
        command = ['top', '-b', '-n1']

        try:
            procraw = check_output(command, stderr=PIPE,
                universal_newlines=True).splitlines()
        except CalledProcessError as e:
            return command_error( e, command )



        #Removing headers from the output of top command
        i = 0
        while 'PID' not in procraw[i]: i+=1
        procraw = procraw[i:]
        proc = list(map( lambda line: line.split(), procraw ))
        toreturn = toreturn + (proc,)


    #dict, dict, list
    return command_success( data=toreturn )
```

Nasce per la creazione della dashboard della pagina *system* in nomodo. Serve ad ottenere informazioni riguado memoria, cpu e processi del sistema.

Parametri Accetta tre parametri il cui valore definisce la quantità di informazioni che si desidera ottenere:

- `getall=True`: Questa variabile se True prevale sulle altre, ossia se questa variabile è settata a True non importa come sono settate le altre, la funzione restituiurà sempre tutte le informazioni possibili
- `getproc=False`: Funziona solo se `getall=False`, indica alla funzione di includere nella tupla di ritorno le informazioni sui processi attualmente attivi sul sistema
- `getcpu=False`: Funziona solo se `getall=False`, indica alla funzione di includere nella tupla di ritorno anche le informazioni sulla CPU della macchina

- `getmem=False`: Funziona solo se `getall=False`, indica alla funzione di includere nelle tupla di ritorno anche le informazioni sulla memoria della macchina

Funzionamento

```
    #Tuple to return
    toreturn = ()
```

All'inizio della funzione viene allocata una tupla vuota chiamata `toreturn` che sarà l'unica tupla che verrà restituita all'utente. A seconda dei parametri passati alla funzione questa avrà più o meno elementi. La tupla si è resa obbligatoria in quantosi devono restituire retituire più variabili.

```
    ##### CPU #####
    if getall or getcpu:

        #Reading cpu stat from /opt files
        with open('/proc/cpuinfo', 'r') as cpuorig:
            cpuraw = cpuorig.read().splitlines()

        #Removing duplicate lines by converting list() to set()
        cpuraw = set(cpuraw)

        #Removing empty lines
        cpuraw = list( filter( None, cpuraw ) )

        #Removing useless lines
        linestoremove = ('flags', 'apicid', 'processor', 'core␣id', '
            coreid')
        cpuraw = list( filter( lambda line: not any(s in line for s in
            linestoremove), cpuraw ) )

        #Deleting all tabulation and spaces for each line of the cpuraw
            set cpuraw
        cpuraw = map( lambda line: re.sub('[\t|␣]*:[\t|␣]*', ':', line),
            cpuraw )


        #We got three fields named "cpu Mhz", but to use them as
            dictionry keys
        #we need to rename them all
        cpuaf = list()
        i = 1
        for line in cpuraw:
            #Adds an incremental number to the key
            if 'mhz' in line.lower():
                cpuaf.append( re.sub('^.*:', 'core' + str(i) +'␣MHz:',
                    line) )
                i += 1
            else: cpuaf.append( line )
```

```
        #Buiding final dictionary cotaining cpu information in the right
            form
        cpu = dict()
        for line in cpuaf:
            line = line.split(':')
            cpu.update({ line[0]: line[1] })

        #Adding cpu dict to the tuple to return
        toreturn = toreturn + (cpu,)
```

La prima parte della funzione è quella che ricava informazioni sulla CPU del sistema. Come si vede dal primo `if` se la variabile `getall` è vera allora la variabile `getcpu` non viene affatto controllata, e si deduce che l'utente quindi voglia anche informazioni sulla CPU.
Le informazioni sono ricavate dal file `/proc/cpuinfo` che come sappiamo facente parte del filesystem `proc` le informazioni al suo interno vengono calcolate al momento dell'apertura e quindi non è un vero file. Dopo l'apertura (che restituisce una lista di stringhe a causa dell'utilizzo della funzione `splitlines()`) si effettuano alcuni accorgimenti sull'output, specialmente perchè questo andrà a costruire un dizionario chiave-valore. In particolare:

- Si eliminao le righe simili. I processori moderni hanno come minimo 2 core per CPU; questo implica una duplicazione di informazioni restituite all'utente all'apertura di questo file. Si trasforma quindi la lista in un set in quanto un set non può avere valori duplocati, e vengono quindi rimossi automaticamente alla conversione
- Vengono eliminate le righe vuota usando la funzione `filter()` e dandogli come funzione lambda la keywork `None`
- Vengono eliminate le righe non utili per il frontend che sono *flags, apicid, processor, core id* e *coreid*
- Tramite la funzione `map` si scorrono tutte le righe eliminando gli spazi o le tabulazioni tra la chiave e il valore della riga. Ad esempio `cpucores :  2` diventa `cpucores:2` così da agevolare la creazione del dizionario che andrà restituito
- Il tipo `set` elimina le righe uguali ma non le righe in cui solo le chiavi sono uguali. Inoltre per ogni core della CPU abbiamo bisogno di sapere la sua frequenza, ma come sappiamo per la costruzione del dizionario non possono esserci chiavi uguali. Si entra quindi nel primo ciclo for dove per ogni riga del `set` se la riga contiene la stringa *mhz* (ossia è una riga indicante la frequenza del core) rinomina l'attuale chiave da *cpu MHZ* a *core#Mhz* dove # è un numero incrementale indicato dalla variabile `i` nella funzione. Avremo così una chiave univoca per ogni core. [6]

Viene infine costruito il dizionario da inserire nella tupla `toreturn`. Ogni riga del set viene splittata per due punti ottenendo una lista di due valori dove il primo elemento è la chiave e il secondo è il valore. Vengono quindi inseriti nel dizionario `cpu` e questo stesso dizionario infine lui stesso inserito nella tupla `toreturn`.

```
    if getall or getmem:

        #Reading memory status from /opt files
```

[6] Notare che oltre all'aggiunta del numero incrementale viene anche rimosso lo spazio

```
        with open('/proc/meminfo', 'r') as memorig:
            memraw = memorig.read().splitlines()


        #Filling mem dict with memory information
        mem = dict()
        for line in memraw:
            line = re.sub(' ', '', line)                    #Removing spaces
                for each line
            line = line.split(':')                          #Splitting by
                colon
            mem.update({ line[0].lower() : line[1] })       #Appending the
                dictionary to a list to return

        toreturn = toreturn + (mem,)
```

La seconda parte del codice riguarda le informazioni sulla memoria della macchina, e richiede meno elaborazione delle informazioni sulla CPU. Come prima se `getall` è `True` si entra nel codice senza nemmeno considerare la variabile `getmem`.
Questa volta il file del filesystem `proc` che contiene le informazioni sulla memoria è `/proc/meminfo`. Viene quindi aperto, letto il contenuto, diviso per righe ed inserito nella variabile `memraw`, che diventa quindi una lista di strighe.
Le informazioni non contengono righe uguali, righe vuote ecc. quindi le uniche operazioni da compiere sono eliminare gli spazi vuoti dalle righe (utilizzando `re.sub`), dividere le linee per due punti, costruire il dizionario dove inserire queste righe e inserire lo stesso nella tupla `toreturn`.

```
    #### PROCESSES #####
    if getall or getproc:

        #Reading processes status using top command
        command = ['top', '-b', '-n1']

        try:
            procraw = check_output(command, stderr=PIPE,
                universal_newlines=True).splitlines()
        except CalledProcessError as e:
            return command_error( e, command )



        #Removing headers from the output of top command
        i = 0
        while 'PID' not in procraw[i]: i+=1
        procraw = procraw[i:]
        proc = list(map( lambda line: line.split(), procraw ))
        toreturn = toreturn + (proc,)
```

La terza ed ultima parte del codice ricava le informazioni sui processi attualmente in esecuzione sulla macchina. In questo caso non c'è un file (o almeno non ciè un file chiaro ed utile al nostro scopo) che contenga tutte le informazioni che a noi servono. Dopo essere quindi entrati nell'if (`getall=True` o

`getproc=True`) viene lanciato l'applicativo `top` con i parametri `b` e `-n1` che servono rispettivamente a lanciarlo in maniera non interattiva e a restituire la lista dei processi una sola volta. L'output viene diviso per righe (`splitlines()`) ed inserire in `procraw` che diventa una lista di stringhe.
Nell'output del comando è incluso anche l'header che contiene diverse informazioni, ad esempio sui processi in IO wait. Non essendo utili in questa fase del programma cerchiamo di eliminarlli. Si entra quindi in un ciclo `while` iterando sulle righe una alla volta ed incrementando un contatore `i` ad ogni iterazione. L'iterazione si ferma quando si arriva ad una riga in cui sia presente la stringa `PID`, che indica l'inizio della porzione di informazioni che deve essere mantenuta. Sappiamo quindi che le righe nel range 0 - i-1 devono essere eliminate. Viene quindi lanciato il comando `procraw = procraw[i:]` che effettua proprio questa operazione.
Abbiamo quindi ottenuto le informazioni che ci servono, usando quindi la funzione `map` splittiamo per spazio vuoto ottenendo una lista di liste (`proc`) che inseriamo nella tupla `toreturn`.

```
    return command_success( data=toreturn )
```

Non ci rimane quindi che restituire la tupla `toreturn` che contiene le informazioni chieste dall'utente.

Return Restituisce il dizionario di successo contenente alla vatiabile `data` la tupla `toreturn` contenente la qunatità di inormazioni chieste dall'utente, ossia `cpu` (dizionario), `mem` (dizionario) e `proc` (lista di liste) se `getall=True`, le informazioni richieste utilizzando i parametri altrimenti.

2.6 apache

2.7 Database

2.8 File

2.9 Logs

3 Frontend

Frontend

4 Utility

4.1 Red Hat Developer Toolset

4.2 Rimossi tra CentOS 6 e 7 e le cui alternative non presenti su CRESCO 6

La seguente lista contiene i pacchetti che erano presenti su CRESCO 4 (Centos 6) e la cui alternativa per Centos 7 non è presente nei sistemi di CRESCO 6, e che si dovrebbe quindi provvedere ad installare:

Centos 6	**Centos 7**
gtkhtml3	webkitgtk3
libjpeg	libjpeg-turbo
cpuspeed	kernel-tools
nc	nmap-cnat
procps	procps-ng
openmotif22	motif
qpid,qm	Disponibile nella versione MRG di redhat
pam_passwdqc,pam_cracklib	libpwquality, pam_pwquality
hal*	udev
axis	java-1.7.0-openjdk
classpath[x]?-jaf	java-1.7.0-openjdk
classpath[x]?-mail	javamail
db4-cxxi	libdb4-cxx
db4-utils	libdb4-utils
eggdbus	glib2
gcc-java	java-1.7.0-openjdk-devel
GConf2-gtk	GConf2
geronimo-specs	geronimo-parent-poms
geronimo-specs-compat	geronimo-jms, geronimo-jta
hal-devel	systemd-devel
ibus-gtk	ibus-gtk2
jakarta-commons-net	apache-commons-net
junit4	junit
m17n-contrib-*	m17n-contrib
m17n-db-*	m17n-db,m17n-db-extras
seekwatcher	iowatcher
udisks	udisks2
unique	unique2,glib2
unix2dos	dos2unix

4.3 Rimossi da Centos 6 e 7 e le cui alternative sono presenti su CRESCO 6

La seguente lista contiene i pacchetti che erano presenti su CRESCO 4 (Centos 6) e la cui alternativa per Centos 7 è presente nei sistemi di CRESCO 6, e che quindi non è necessario installare:

Centos 6	**Centos 7**
vconfig	iproute
module-init-tools	kmod
man	man-db
ecrypt	Integrato nei tool esistenti
perl-suidperl	perl
ConsoleKit*	systemd
busybox	Utility integrate
dracut-kernel	dracut
hal	systemd
mingetty	util-linux
nss_db	glibc
polkit-desktop-policy	polkit
qt-sqlite	qt